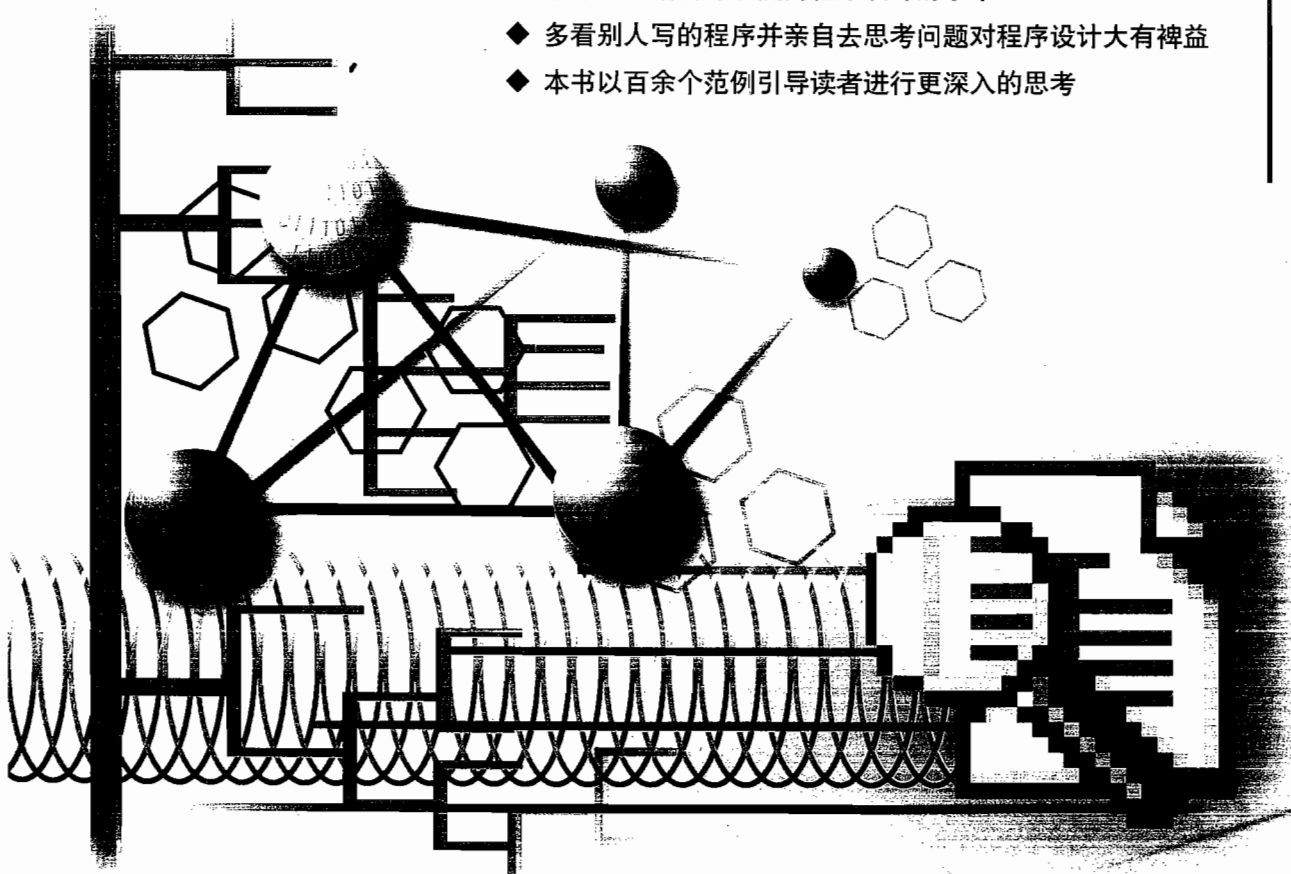


数据结构 (C语言版)

- ◆ 数据结构是前人在思索问题时想出的解决方法
- ◆ 学习数据结构可以提高程序设计的水平
- ◆ 多看别人写的程序并亲自去思考问题对程序设计大有裨益
- ◆ 本书以百余个范例引导读者进行更深入的思考



黄国瑜 叶乃菁 编著



清华大学出版社
<http://www.tup.tsinghua.edu.cn>



数 据 结 构

(C 语言版)

黄国瑜 叶乃菁 编著

清 华 大 学 出 版 社

(京)新登字 158 号

内 容 简 介

数据结构包含以下两方面的内容：一是用合适的运算法则来规划程序流程，二是采用简洁的数据结构来表示程序中的数据和变量。

本书以 C 语言为程序设计语言，采用条列式的叙述方式，引导读者循序渐进地掌握堆栈结构、队列结构、树状结构、字符串结构，以及递归设计、排序设计和查找设计等程序设计。全书文字浅显易懂，程序示例简洁明了，是程序设计人员的上乘参考书。

本书繁体字版名为《资料结构》，由文魁资讯股份有限公司出版，版权归黄国瑜，叶乃菁所有。本书简体字中文版由文魁资讯股份有限公司授权清华大学出版社独家出版。未经本书原出版者和本书出版者书面许可，任何单位和个人不得以任何形式或任何手段复制或传播本书的部分或全部。

北京市版权局著作权合同登记号：图字 01-2000-4119 号

版权所有，翻印必究。

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

书 名：数据结构(C 语言版)

作 者：黄国瑜 叶乃菁

责任编辑：张彦青

出 版 者：清华大学出版社(北京清华大学学研大厦，邮编 100084)

<http://www.tup.tsinghua.edu.cn>

印 刷 者：北京市清华园胶印厂

发 行 者：新华书店总店北京发行所

开 本：787×1092 1/16 印张：28.25 字数：790 千字

版 次：2001 年 8 月第 1 版 2001 年 11 月第 2 次印刷

书 号：ISBN 7-302-04509-7/TP·2665

印 数：5001~10000

定 价：32.00 元

数据结构与程序

R.C.L. 张宇/张宇 张宇

本页已使用福昕阅读器进行编辑。
福昕软件(C)2005-2008, 版权所有,
仅供试用。



许多人都会设计程序，但并非每位程序设计者都能写出好的程序。怎样才能编写出好的程序呢？我们强调两方面，一方面是要找出合适的算法来设计程序的流程，另一方面则是采用简洁适用的数据结构来表示程序中的数据和变量。而“数据结构”含括了这两方面的概念，故“数据结构”为程序设计者最重要的基本要求。

“数据结构”这门学科是前人在程序设计方面积累的程序设计经验，“学会基础程序设计，你只能解决程序设计中三成的问题；学会数据结构，你却能解决程序设计中八成的问题。”因此，“数据结构”是信息科学中被认为最重要的学科之一。但是“数据结构”并不容易学习，所以常让初学者望而却步。

为了减少读者在学习“数据结构”上的恐惧，本书在每章节中都附有详细的说明和程序范例，藉此带领读者一步一步地学习“数据结构”，并希望读者在学习本书的过程中能够获得最大的收益。

在本书中我们采用条列步骤的方式，引导读者循序渐进地学习各种不同的数据结构，还使用图形来表示每一步骤的处理操作，再加上浅显易懂的文字说明及程序范例，让读者可以轻轻松松地了解数据结构的概念。

虽然本书在编写过程中力求完美，仍难免有错漏之处，希望各位不吝指正。

WANT 工作室 黄国瑜
Cherish 工作室 叶乃菁

{ 线性结构
非线性结构

目 录

第 1 章 数据结构的基本概念	1
1.1 何谓数据结构	2
1.2 算法与伪码	2
1.3 程序结构化与设计风格	4
1.4 程序分析的方法	8
1.5 时间复杂度分析	10
1.6 渐近式表示法	13
1.6.1 时间复杂度各类等级	13
1.6.2 渐近式表示法	14
1.7 递归式的复杂度计算	16
第 2 章 数组	21
2.1 何谓数组	22
2.2 一维数组	22
2.3 一维数组的使用	24
2.4 一维数组的存取	26
2.5 一维数组的遍历	28
2.6 一维数组的高级应用	29
2.7 二维数组	33
2.8 数组表示法	37
2.9 特殊类型的数组	42
2.9.1 稀疏数组	42
2.9.2 上三角数组	44
2.9.3 下三角数组	49
第 3 章 链表	55
3.1 何谓链表	56
3.2 单链表的建立	56
3.2.1 单链表内节点的配置	56
3.2.2 单链表内节点的释放	58

3.2.3 单链表的建立与释放	59
3.2.4 单链表的查找	63
3.3 单链表的基本处理	65
3.3.1 单链表内节点的插入	65
3.3.2 单链表内节点的删除	70
3.3.3 单链表的反转	74
3.3.4 单链表的链接	79
3.3.5 单链表的比较	82
第 4 章 堆栈	87
4.1 何谓堆栈	88
4.2 用数组仿真堆栈	88
4.3 用链表仿真堆栈	92
4.4 表达式表示法	96
4.5 中序表达式的表示法及计算	97
4.6 前序表达式的表示法及计算	102
4.7 后序表达式的表示法及计算	105
4.8 表达式的转换	108
第 5 章 队列	115
5.1 何谓队列	116
5.2 用数组仿真队列	116
5.3 用链表仿真队列	121
5.4 环状队列	125
5.5 双向队列	130
5.5.1 输入限制性双向队列	130
5.5.2 输出限制性双向队列	134
第 6 章 递归	139
6.1 何谓递归	140
6.2 函数调用与参数传递	142
6.3 数学问题	146
6.3.1 阶乘问题	147
6.3.2 最大公因子问题	148
6.3.3 费氏级数问题	149
6.3.4 组合公式	151
6.4 河内塔问题	153
6.5 N 皇后问题	158
6.6 迷宫问题	166

第 7 章 基础树状结构	175
7.1 何谓树状结构	176
7.1.1 何谓树	176
7.1.2 树的相关名称及意义	176
7.2 二叉树	177
7.2.1 何谓二叉树	177
7.2.2 二叉树和树的比较	178
7.2.3 二叉树的相关特色	178
7.3 二叉树表示法	179
7.3.1 二叉树数组表示法	180
7.3.2 二叉树结构数组表示法	183
7.3.3 二叉树链表表示法	187
7.4 二叉树的遍历	190
7.4.1 二叉树的前序遍历	190
7.4.2 二叉树的中序遍历	193
7.4.3 二叉树的后序遍历	196
7.5 二叉树的建立(递归法)	199
7.6 二叉树的查找	201
7.6.1 何谓二叉查找树	201
7.6.2 二叉树的查找方式	202
7.7 二叉树的节点删除	205
7.7.1 节点无左子树, 无右子树	205
7.7.2 节点有左子树, 无右子树	206
7.7.3 节点无左子树, 有右子树	207
7.7.4 节点有左子树, 有右子树	207
7.8 二叉树的复制	212
7.9 二叉树的比较	214
7.10 二叉树的映像	218
7.11 一般树转二叉树	221
7.12 引线二叉树	223
7.13 二叉树的应用(表达式)	229
第 8 章 排序	235
8.1 何谓排序	236
8.1.1 排序的意义	236
8.1.2 排序的特性——稳定性与不稳定性	236
8.1.3 排序的分类	237
8.2 内部排序法——交换式排序	237

8.2.1 冒泡排序法	237
8.2.2 快速排序法	242
8.3 内部排序法——选择式排序	247
8.3.1 选择排序法	247
8.3.2 累堆排序法	251
8.4 内部排序法——插入式排序	258
8.4.1 插入排序法	259
8.4.2 谢耳排序法	262
8.4.3 二叉树排序法	265
8.5 外部排序——合并排序法	268
8.6 排序法的效率比较	273
第9章 查找	275
9.1 何谓查找	276
9.2 线性查找	276
9.3 折半查找	280
9.4 费氏查找	285
9.5 插补查找	290
9.6 杂凑查找	299
9.6.1 杂凑函数	299
9.6.2 杂凑碰撞解决法	303
9.6.3 杂凑查找	307
9.7 二叉查找树	314
第10章 高级链表	319
10.1 循环链表	320
10.1.1 循环链表的建立与释放	320
10.1.2 循环链表内节点的插入	324
10.1.3 循环链表内节点的删除	329
10.2 双链表	334
10.2.1 双链表的建立与释放	334
10.2.2 双链表的插入	337
10.2.3 双链表的删除	343
第11章 字符串结构	353
11.1 字符串的声明	354
11.2 字符串的基本 I/O	355
11.3 字符串的传递方式	356
11.4 字符串的基本处理	357
11.4.1 字符串的长度计算: Strlen(char *s)	358

11.4.2	字符串的复制——Strcpy(char *s1,char *s2).....	359
11.4.3	字符串的结合——Strcat(char *s1,char *s2).....	360
11.4.4	字符串的取代——Strrep(char *s1,char *s2,int pos).....	361
11.4.5	字符串的插入——Strins(char *s1,char *s2,int pos).....	363
11.4.6	字符串的删除——Strdel(char *s1,int pos,int len).....	364
11.5	字符串的高级处理	366
11.5.1	字符串的比较——Strcmp(char *s1,char *s2).....	366
11.5.2	抽取子字符串——Substr(char *s1,int pos,int len).....	367
11.5.3	字符串的比较	369
11.5.4	字符串的分割	372
11.5.5	常用的字符串函数	373
11.6	字符串转换数值的应用	374
第 12 章	图形结构	377
12.1	何谓图形结构	378
12.1.1	无向图形	378
12.1.2	有向图形	379
12.1.3	完全图形	379
12.1.4	子图形	379
12.1.5	路径	379
12.1.6	简单路径	380
12.1.7	回路	380
12.1.8	连通顶点	380
12.1.9	连通图形	380
12.1.10	连通单元	380
12.1.11	强连通顶点	381
12.1.12	强连通图形	381
12.1.13	强连通单元	381
12.2	图形的表示法	381
12.2.1	邻接数组表示法	381
12.2.2	邻接列表表示法	384
12.2.3	多重邻接列表表示法	389
12.2.4	加权边的图形	394
12.3	图形的查找	395
12.3.1	深度优先法	395
12.3.2	广度优先法	398
12.3.3	连通组件	403
12.4	生成树问题	403
12.4.1	生成树	403

12.4.2 最小生成树	405
12.4.3 Kruskal 算法	405
12.4.4 Prims 算法	411
12.5 最短路径问题	415
附录 A ASCII 码	425
附录 B 习题解答	429

数据结构的基本概念

第 1 章

- ◆ 何谓数据结构
- ◆ 算法
- ◆ 程序结构化与设计风格
- ◆ 程序分析的方法
- ◆ 时间复杂度分析
- ◆ 渐近式表示法
- ◆ 递归式的复杂度计算

1.1 何谓数据结构

程序设计正如写篇英语作文一样，每个人都懂得英语的语法和基本句型，可是同样的作文题目，每个人写出的作文，却风格各异。同样的道理，程序设计也是一样，程序员都懂得程序的语法与语意，可是相同功能的程序，由不同程序员写出来的程序解法也会不一样。有人写出来的程序，程序效率很高，有人却用最复杂的方法来解决一个简单的问题。

当然程序设计的水平，绝非看了几本程序设计的书，就可以功力大增的。“师父领进门，修行看个人”，同样看完3本程序设计书的两个人，程序设计水平高的，一定是愿意亲自思索问题、多练习的人。记得刚学程序设计时，常听人说程序设计的水平要想提高，除了练好基本功夫外，最重要的还是多看别人写的程序，多亲自尝试去思考问题。从看别人写的程序中，我们可以发现出更多更有效率的解决方法，从亲自思考问题的过程中，我们更可以了解问题的解决方法常常不只一个，如何运用以前解决问题的经验，来解决更复杂更深入的问题，才是最有效的。

而“数据结构”，这门信息科学上的学问，正是前人在思索问题的过程中，所想出的解决方法。一般而言，在学习程序设计一段时间后(练好程序设计的基础以后)，学习“数据结构”便可以大大提高程序设计上水平。如果你只学会了程序设计的语法和语意，你只能解决程序设计上三分之一的问题，而且运用的方法，并不是最有效率的。但如果你学会了数据结构的概念，你却能在程序设计上，运用最有效的方法来解决八成以上的问题。

在程序语言中，“数据类型”是指程序语言中变量所能表示并存储的数据种类，“数据实体”则是指在一种数据类型中的所有可能元素的集合。而“数据结构”，大致上说来，就是数据实体中元素之间的关系，包括数据的表示法和运算。例如：链表(以后章节中会有详细的介绍)，是指一个将具有数据内容和指向下一笔数据指针的节点串连而成的数据结构。堆栈是指各元素具有先进后出(First In Last Out)特性的数据结构，当然还有很多种数据结构与问题的解法。本书会在后面的章节中，介绍几种常见的数据结构和问题的解决方法，帮助读者了解数据结构。

希望读者除了详读本书以外，还能试着用自己的方法，来解决本书所附的范例和习题，相信这会使程序设计水平有很大的提高。

1.2 算法与伪码

在开始这节以前，我们先定义出什么是“算法(Algorithms)”？算法是指为了完成某项特定工作所设计出一连串用来说明工作是如何被完成的步骤。所有的算法都必须满足下列几个条件：

- 输入：不具有输入数据或具有多个输入数据。
- 输出：具有一个以上的结果输出。
- 定义明确(Definiteness)：每一个步骤的语句必须很明确。
- 有限的步骤(Finiteness)：按照算法所描述的步骤执行，在有限的步骤内，一定会结束。
- 有效率的步骤(Effectiveness)：算法中的每一个步骤必须是基本的指令(即使是用纸和笔也可以完成计算)。

简单的说，算法就是一个具有次序、步骤清楚，最后一定会有执行结束的可执行步骤。程序与算法不同的地方在于上述的第4点，算法必须是一个可执行完的步骤，而程序却允许存在死循环(一个具有死循环的程序，我们也称为程序)。

用来写算法的方式有好几种，一般而言，我们可以分为下列4种方式：

- 条列式的步骤:

以条列式的步骤来描述解决问题的方法。

例如: 运用顺序查找来查找数据中某一个特定值。

步骤 1: 输入数据和欲查找值。

步骤 2: 查找数据中第一项。

步骤 3: 如果数据全都查找过但未能查找到欲查找值, 表示没有查找到数据。

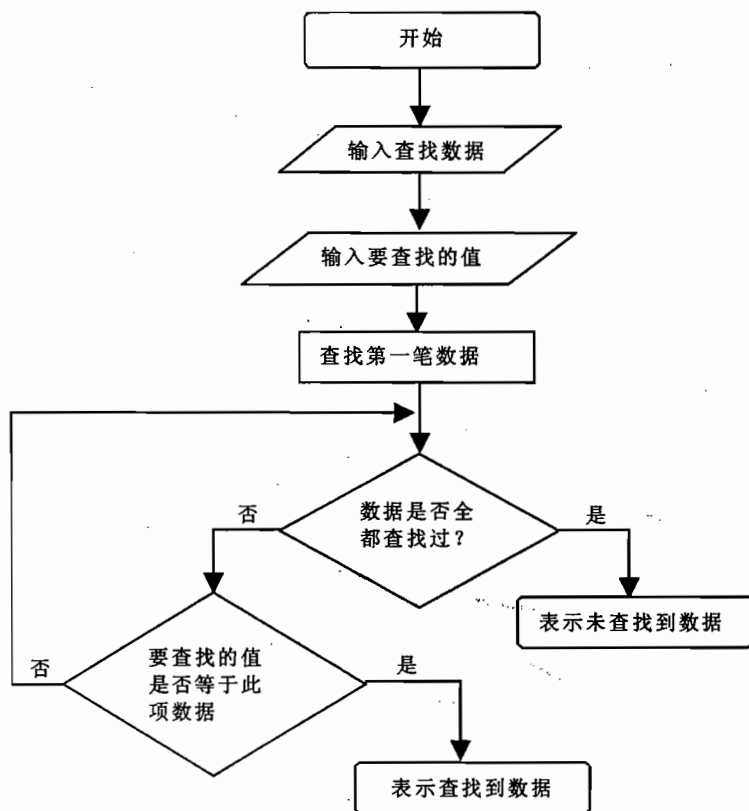
步骤 4: 如果欲查找值等于此项数据, 表示查找到数据。

步骤 5: 如果欲查找值不等于此项数据, 查找下一项数据, 回到第 3 步继续执行。

- 流程图(Flow Chart):

以图形符号来描述解决问题的方法。仅适用于小问题, 不过现在已很少用。

例如: 运用顺序查找来查找数据中某一个特定值。



- 伪码(Pseudo Code):

以夹杂程序语法和自然语言(如: 中文、英文)的形式来描述解决问题的方法。

例如: 运用顺序查找法来查找数据中某一个特定值。

```

Procedure Sequential_Search(Data, KeyVal)
  设 I 为 1;
  while ( )
  {
    if ( 欲查找值等于 Data[I] )
      printf("查找到数据。");
    else if ( I > 数据个数 )           // 数据全查找完
      printf("未能查找到数据。");
    I++;
  }

```

}

● 程序语句:

直接以程序语法来描述解决问题的方法。

例如: 运用顺序查找法来查找数据中某一个特定值。

```

01      /* ===== Program Description ===== */
02      /* 程序名称: s_search.c */
03      /* 程序目的: 设计一个顺序查找的程序。 */
04      /* Written By Kuo-Yu Huang. (WANT Studio.) */
05      /* ===== */
06      int Data[20] = /* 输入数据数组 */
07      {
08          1, 7, 9, 12, 15,
09          16, 20, 32, 35, 67,
10          78, 80, 83, 89, 90,
11          92, 97, 108, 120, 177};
12      int Counter = 1; /* 查找次数计数变量 */
13
14      /*-----*/
15      /*顺序查找 */
16      /*-----*/
17      int Seq_Search(int Key)
18      {
19          int i; /* 数据索引计数变量 */
20
21          for ( i=0 ; i<20 ; i++ )
22          {
23              printf("[%d]",Data[i]); /* 输出数据 */
24              if ( Key == Data[i] ) /* 查找到数据时 */
25                  return 1;
26              Counter++; /* 计数器递增 */
27          }
28          return 0;
29      }
30
31      /*-----*/
32      /*主程序 */
33      /*-----*/
34      void main ()
35      {
36          int KeyValue; /* 欲查找数据变量 */
37
38          printf("Please enter your key value : ");
39          scanf("%d",&KeyValue); /* 输入欲查找值 */
40
41          if ( Seq_Search(KeyValue) ) /* 调用顺序查找子程序 */
42              /* 输出查找次数 */
43              printf("\nSearch Time = %d\n",Counter);
44          else
45              printf("No Found!!\n"); /* 输出没有找到数据 */
46      }

```

1.3 程序结构化与设计风格

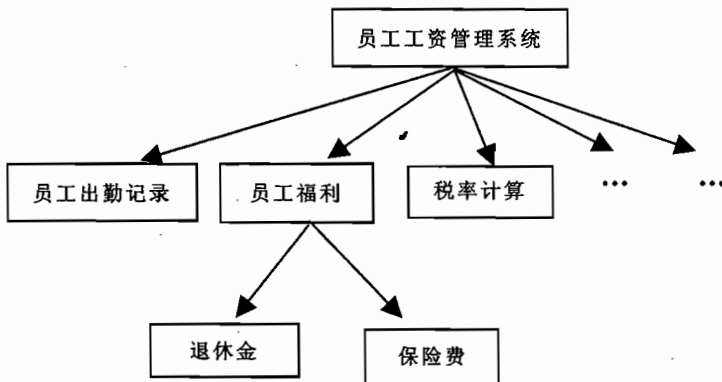
随着程序所要解决的问题愈来愈多时,程序的代码量也会愈来愈大,本来我们只是写写一、两百行的程序,可是当所要解决的问题很大时,程序可能就需要上万行才能解决该问题。一个小程序,在程序写完几个月后,我们再重新阅读时,很轻易的就能再了解当初所写的程序,但是当程序量很大时,如果没有一

个特定的程序编写方式,以后重新阅读时,一定困难重重。这一节我们所要说明的正是程序结构化与设计风格,希望藉此帮助读者建立程序的设计风格。

其实程序员所写的程序并不是一次就能把问题完整的解决,程序员花在程序维护和除错上的时间常常多于程序编写的时间。一个程序如果没有一定的风格,不仅别人看不懂,最后可能连自己也会看不懂,最糟的情况可能还得重写一次,因为重写的时间可能要比看懂原程序的时间短。的确,一份设计风格明确、简洁易懂的程序,可节省程序员在维护、除错上所花的时间,并有助于程序以后的更新与发展。

首先,我们来谈程序的结构化。程序结构化最重要的概念是“模块化(Modularity)”,所谓模块化就是将原来的大问题区分成几个小问题,再从解决小问题的过程中,组合成大问题的解法。对于解决小问题的小程序或许一个程序员就能胜任,但是一个大问题,常常是一群程序员共同努力的成果。所以模块化的概念,可将每个小模块分工给各小组的程序员来开发,最后再组合成一个完整的软件。

例如:一套员工工资管理系统,可能分为几个小模块,如下图所示,每一个方块代表一个模块。



区分模块以后程序员只要分头去进行各模块的程序编写工作,最后即可完成一套完整的员工工资管理系统。

在此介绍读者两种程序设计发展的方式:

- 由上而下设计(Top-Down Design):

由上而下设计法的第一步通常是产生较简短的解法,再一步一步地修正前一个步骤所产生的解法,将前一个大模块区分成更小的模块,解决小模块以组合出大模块的解法。

- 由下而上设计(Bottom-Up Design):

由下而上设计法则是先列举出一个大模块中,各个个别的小模块,分别建立出各个模块的解决方案,再应用于大模块中。这个概念随着对象导向程序设计而逐渐受欢迎,因为在对象导向程序设计中,常常是建立一个独立的对象,再通过小对象来完成大问题。

除了程序的结构化以外,程序的编写风格也是重点,笔者在此提供几个程序编写上的方法,以帮助读者建立起良好的编写风格。

1. 注释

一份良好的程序,除了程序本身外,最重要是要有一份完整的程序说明文件。一份没有注释的程序,宛如是一部天书,常常会让负责维护的程序员,搞不懂原设计者的设计目的。但一份注释完整的程序,除了自己阅读和除错上的方便外,更容易让人了解你的程序,并赞叹你在程序设计上的巧思与创意。

我们再举以前所提的顺序查找法为例:

```

01      /* ===== Program Description ===== */
02      /* 程序名称:  s_search.c                      */
03      /* 程序目的:  设计一个顺序查找的程序。        */
04      /* Written By Kuo-Yu Huang. (WANT Studio.)    */
05      /* ===== */
  
```



```

06  int Data[20] = /* 输入数据数组 */
07  {   1,   7,   9,  12,  15,
08  16,  20,  32,  35,  67,
09      78, 80,  83,  89,  90,
10      92, 97, 108, 120, 177};
11  int Counter = 1; /* 查找次数计数变量 */
12
13  /* ----- */
14  /* 顺序查找 */
15  /* ----- */
16  int Seq_Search(int Key)
17  {
18      int i; /* 数据索引计数变量 */
19
20      for ( i=0 ; i<20 ; i++ )
21      {
22          printf("[%d]",Data[i]); /* 输出数据 */
23          if ( Key == Data[i] ) /* 查找到数据时 */
24              return 1;
25          Counter++; /* 计数器递增 */
26      }
27      return 0;
28  }
29
30  /* ----- */
31  /* 主程序 */
32  /* ----- */
33  void main ()
34  {
35      int KeyValue; /* 欲查找数据变量 */
36
37      printf("Please enter your key value : ");
38      scanf("%d",&KeyValue); /* 输入欲查找值 */
39
40      if ( Seq_Search(KeyValue) ) /* 调用顺序查找子程序 */
41          /* 打印查找次数 */
42          printf("\nSearch Time = %d\n",Counter);
43      else
44          printf("No Found!!\n"); /* 打印没有找到数据 */
45  }

```

从此程序中，我们可以发现在程序的开头前 5 行：

```

/* ===== Program Description ===== */
/* 程序名称: s_search.c */
/* 程序目的: 设计一个顺序查找的程序。 */
/* Written By Kuo-Yu Huang. (WANT Studio.) */
/* ===== */

```

我们在注释中写上程序名称、程序目的及作者。程序名称注明了这个程序可以在哪一个文件中找到，往后在查阅或修改时，就可以马上找到该文件进行修改。当然在修改时，如果能在程序开头再注明修改的时间、内容及修改者就更好了。程序目的则注明了这个程序的功用，帮助下一次运行时，无需重新看完程序，即可了解其功用。而程序作者部分，则是帮助以后想看懂这个程序或想修改的人，有问题时可以求助于原作者。

第 13~15 行是子程序的注释格式。

```

/* ----- */
/* 顺序查找 */
/* ----- */

```

我们在注释中，只需注明这个子程序功用即可，因为程序的主要功用已在程序开头的批注中说明过了。

第6、11、18、35行则是程序中变量的注释格式:

```
int    Counter = 1;    /* 查找次数计数变量 */
```

如果可以的话,最好是每一个变量用一行来声明,有人习惯在一行中声明多个变量,如下列这种方式:

```
int    Index,Counter,key,I,j;
```

除非这几个变量是同一功用的变量(如循环计数变量),否则的话,这种变量声明方式,不仅在注释上不易,在阅读上更显得杂乱。

其余的程序批注,可选择在重要的程序语句后面加上一个注释内容。

2. 变量命名

变量是程序中用于记录输入值或输出结果的一个内存位置,所以变量所象征的意义正如同该变量在程序中所代表的意义。如果我们想要写一个计算学生成绩的程序,程序中需要用户输入学生学号、语文成绩、英语成绩、数学成绩,最后再计算出3科的平均成绩。此时如果程序中的变量声明为:

```
int    X;
int    A,B,C;
int    D;
```

我们没有人会看的懂几个变量所代表的意义,就算在程序以初就注明 X 是代表学生学号、A 代表语文成绩、B 代表英语成绩、C 代表数学成绩, D 代表3科平均成绩,在程序中,也会很容易忘了什么变量代表什么意义。如果把这4个变量声明为:

```
int    StudentNum;    /* 学生学号 */
int    Chinese;       /* 语文成绩 */
int    English;       /* 英语成绩 */
int    Math;          /* 数学成绩 */
int    Average;       /* 3科平均 */
```

这样是不是比较清楚易懂呢?因为变量的声明通常会在某一个特定的区域(如:程序开头),如果在变量以后再加上一些批注说明,这样在编写或修改程序以际,变量声明的区域就像是一个小字典,提供给我们所有在此程序中的输入和输出信息。

3. 程序缩排

在程序中经常会用到一些条件式语句或循环结构,在这个语句当中,包含了 C 语言中区块(Block),也就是一群单一语句的集合,通常是以“{”及“}”来划分。“{”表示区块的开始,“}”表示区块的结束,在区块划分上有两种设计的风格。

一种是把开头的“{”与语句放在同一行,如:

```
for ( i=0 ; i<20 ; i++ ) {
    printf("[%d]",Data[i]);    /* 输出数据 */
    if ( Key == Data[i] )      /* 查找到数据时 */
        return 1;
    Counter++;                /* 计数器递增 */
}
```

另一种是把开头的“{”不与语句放在同一行,如:

```
for ( i=0 ; i<20 ; i++ )
{
    printf("[%d]",Data[i]);    /* 输出数据 */
    if ( Key == Data[i] )      /* 查找到数据时 */
        return 1;
    Counter++;                /* 计数器递增 */
}
```

到底哪一种比较好?并没有绝对的答案。不过笔者较偏好于后者,因为其结构清晰,程序也一目了然。

}

到底哪一种比较好? 并没有绝对的答案。不过笔者较偏好于后者, 因为其结构清晰, 程序也一目了然。

另外程序中最好有缩排, 缩排可以帮助程序员减少除错的时间。缩排通常在一些区块、条件语句、循环结构上产生出层次的关系, 缩排通常以空 2 个空格、4 个空格或 8 个空格。其中以 4 个空格最佳。

4. 段落

程序中除了子程序以外, 还有一些专为某一个目的所写的语句, 这些语句的个数不等, 在编写程序时, 不同的目的语句以间最好插入一个空白行, 再加上批注, 这可使程序有段落的感觉, 在除错时也较容易找出错误。

1.4 程序分析的方法

我们该如何来判断一个程序编写的好坏? 其实每个人编写的程序各有各的风格, 我们无法客观地判断谁编写的程序最好或谁写的不好?

唯一可以客观的判断是“程序运行效率”。一个程序结构的好坏, 关键通常在于该程序是否能以最短的时间及最小的空间来运行出结果。但是“时间”与“空间”常常是“鱼与熊掌”无法兼得, 如果希望以最短的时间来完成该程序, 则常常需要运用更多的存储空间, 来换取最短的执行时间; 如果希望以最小的存储空间来完成该程序, 相反的也必须用较久的时间, 才能完成该程序。

若依分析的时间来区分的话, 程序效率的分析可分为事前预测(Priori-Estimate)和事后测试(Posteriori Testing)。事前预测是程序未运行前, 我们先依程序可能使用的时间与空间来评断其效率, 这也就是本章我们所要学习的重点, 然而程序效率事前预测, 并不是那么的容易。

比如, 有一行程序如下:

```
Number=Number*2
```

这行程序的功用为将变量 Number 乘以 2 后存回变量 Number 中。我们假设程序中每一个指令执行的时间为 t , 则上述程序所需的时间则为 t 。若有另一个程序是以循环的方式执行上述程序 n 次, 则程序所需的时间则为 $n*t$ 。但是每一个指令执行的时间 t , 却常常决定于下列几个因素:

- 机器的速度:

一个程序在 486 计算机上运行的速度绝对会比在 P-III 上运行的速度慢。

- 指令集:

若程序指令集已具备做乘法的指令, 则只需一个步骤即可完成该项乘法工作, 若程序指令集中没有乘法的指令, 则需花更多的步骤才能完成乘法工作。

例如: 已具备乘法的指令的指令集其编译后的指令如下(为了方便读者了解, 所以用汇编语言来表示, 而不是机器语言来表示):

```
MUL    Number, 2           ; 将 Number 和 2 做乘法运算后存回 Number 中
```

若不具乘法指令集, 则其编译后的指令可能如下(Number 连加两次):

```
ADD    Number, Number
ADD    Number, Number
```

- 指令周期:

指令集中每一个指令所需要的运行时间也会影响到程序的效率。

- 编译器:

由不同的编译器所编译出的机器语言不一定相同。同样的一行程序经不同的编译器编译后,

率计数)来作为程序效率分析的方式。

对于下列程序,显然其运行次数为1。

```
...
X = X + 1;
...
```

而下列程序:

```
...
for (I=0; I<n; I++)
    X = X + 1;
...
```

则其运行次数为n。

下列程序:

```
...
for (I=0; I<n; I++)
    for (J=0; J<n; J++)
        X = X + 1;
...
```

如:

```
for (I=0; I<n; I++)
    for (J=0; J<n; J++)
        X = X + 1;
```

则其运行次数为 n^2 。

以下我们就以求费氏级数的程序来说明。

程序实例:

设计一个求费氏级数的程序。

程序构思:

费氏级数值为

0 1 1 2 3 5 8 13 21 34 55 ...

当N大于2时,第N项为第N-1项和第N-2项的和。

$$F_n = F_{n-1} + F_{n-2}$$

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: fibn.c */
03  /* 程序目的: 设计一个求费氏级数的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06
07  /* ----- */
08  /* 主程序 */
09  /* ----- */
10  void main ()
11  {
12      int    Number;          /* 运算数值变量 */
13      int    FibN;            /* 费氏级数变量 Fn */
14      int    FibN1;           /* 费氏级数变量 Fn1 */
15      int    FibN2;           /* 费氏级数变量 Fn2 */
16      int    i;               /* 循环计数变量 */
```

```

13      int    FibN;           /* 费氏级数变量 Fn */
14      int    FibN1;          /* 费氏级数变量 Fn1 */
15      int    FibN2;          /* 费氏级数变量 Fn2 */
16      int    i;              /* 循环计数变量 */
17
18      printf("The Fibonacci Numbers \n");
19      printf("Please enter a number : "); /* 输入数值 */
20      scanf("%d",&Number);
21
22      if ( Number <= 1 )
23          printf("Fibonacci Numbers of %d = 1\n",Number);
24      else
25      {
26          FibN2 = 0;
27          FibN1 = 1;
28          for ( i=2 ; i<=Number ; i++ )
29          {
30              FibN = FibN1 + FibN2;
31              FibN2 = FibN1;
32              FibN1 = FibN;
33          }
34          printf("Fibonacci Numbers of %d = %d\n",Number,FibN);
35      }
36  }

```

运行结果:

```

C:\DS>fibn
The Fibonacci Numbers
Please enter a number : 10
Fibonacci Numbers of 10 = 55

C:\DS>

```

接下来,我们就来讨论这个程序:

1. 当 $Number \leq 1$ 时:
 程序中第 18、19、20、22、23 行各执行一次,共执行 5 个指令,执行次数(Frequency counter)为 5。
2. 当 $Number > 1$ 时:
 程序中第 18、19、20、22、26、27 行各执行一次。共执行 6 个指令。
 程序中第 28 行执行 $Number$ 次。共执行 $Number$ 个指令。
 程序中第 30、31、32 行各执行 $Number-1$ 次。共执行 $3*(Number-1)$ 个指令。
 程序中第 34 行执行一次。共执行 1 个指令。
 所以当 $Number > 1$ 时,共执行 $6+Number+3*(Number-1)+1 = 4 * Number + 4$ 个指令,执行次数为 $4 * Number + 4$ 。

上述的程序范例, $Number = 10$, 所以执行次数为 $4 * 10 + 4 = 44$ 。

1.5 时间复杂度分析

我们将每个程序所花费的运行次数称为该程序的“时间复杂度(Time complexity)”,运用时间复杂度的概念,我们即可判断该程序的执行效率是否良好,也方便对两个程序进行分析与比较。

前一节我们定义了“执行次数(Frequency count)”，但是我们从上节的费氏级数的例子中发现，并不是每次的执行次数都是 $4 * \text{Number} + 4$ 。当 $\text{Number} \leq 1$ 时，执行次数则为 5。所以我们可以发现出执行次数与输入值(Input value)与输入的数据个数(Input size)息息相关。

这一节，我们所要学习的正是如何对一个程序的时间复杂度进行分析，以下我们就以顺序查找的程序来说明。

程序实例：

设计一个顺序查找的程序。

程序源代码：

```

01  /* ===== Program Description ===== */
02  /* 程序名称: s_search.c */
03  /* 程序目的: 设计一个顺序查找的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  int Data[20] = /* 输入数据数组 */
07      {
08          1, 7, 9, 12, 15,
09          16, 20, 32, 35, 67,
10          78, 80, 83, 89, 90,
11          92, 97, 108, 120, 177};
12  int Counter = 1; /* 查找次数计数变量 */
13  /* ----- */
14  /* 顺序查找 */
15  /* ----- */
16  int Seq_Search(int Key)
17  {
18      int i; /* 数据索引计数变量 */
19
20      for ( i=0 ; i<20 ; i++ )
21      {
22          printf("[%d]",Data[i]); /* 输出数据 */
23          if ( Key == Data[i] )
24              Counter i ) /* 查找到数据时 */
25              ++; /* 计数器递增 */
26      }
27      return 0;
28  }
29
30  /* ----- */
31  /* 主程序 */
32  /* ----- */
33  void main ()
34  {
35      int KeyValue; /* 欲查找数据变量 */
36
37      printf("Please enter your key value : ");
38      scanf("%d",&KeyValue); /* 输入欲查找值 */
39
40      if ( Seq_Search(KeyValue) ) /* 调用顺序查找子程序 */
41          /* 输出查找次数 */
42          printf("\nSearch Time = %d\n",Counter);
43      else
44          printf("No Found!!\n"); /* 输出没有找到数据 */
45  }

```

运行结果:

```
C:\DS>s_search
Please enter your key value : 1
[1]
Search Time = 1

C:\DS>s_search
Please enter your key value : 67
[1][7][9][12][15][16][20][32][35][67]
Search Time = 10

C:\DS>s_search
Please enter your key value : 50
[1][7][9][12][15][16][20][32][35][67][78][80][83][89][90][92][97][108]
[120][177]
No Found!!

C:\DS>
```

从上述的例子我们可以看出,当欲查找的数据就在数组中第一笔,我们只需查找一次就可以找到,若欲查找的数据在数组中,我们多花一些查找次数也可以找到该笔数据,如从上例中查找 67,需要花 10 次才能查找到数据。但是若欲查找的数据并不在数组中,我们则需要查找 N 次(假设数据有 N 笔)。同样的,若欲查找的数据在最后一笔,则我们也需要查找 N 次。

从上面的讨论过程中,我们归纳出了上述程序可能的查找次数:

- 最佳状况(Best-Case): 当数据在第一笔时,第一次就找到。
- 最坏状况(Worst-Case): 当数据不存在或数据在最后一笔时,需要 N 次。

最佳状况的讨论,其实就是“最佳状况的时间复杂度(Best-case time complexity)”,最佳状况的时间复杂度所能提供给我们的是该程序最低的时间复杂度。也就是该程序时间复杂度的下限(Lower bound)。我们将上述程序的最佳状况的时间复杂度写做: $B(n) = 1 \in O(1)$ 。

而最坏状况的讨论,就是“最坏状况的时间复杂度(Worst-case time complexity)”,最坏状况的时间复杂度所能提供给我们的则是该程序最高的时间复杂度。也就是该程序时间复杂度的上限(Upper bound)。我们将上述程序的最坏状况的时间复杂度写做: $W(n) = n \in O(n)$ 。

除了上述两种外,还有一般状况的时间复杂度(Every-case time complexity, 记做: $E(n)$)和平均状况的时间复杂度(Average-case time complexity, 记做: $A(n)$),平均状况的时间复杂度可让我们清楚判断,当我们输入多笔不同的数据时,其平均的时间复杂度。

例如上述的顺序查找的平均状况的时间复杂度,假设有 n 笔数据,则:

$$\text{平均查找次数: } \frac{n * (n+1)}{2} = \frac{n+1}{2}$$

$$A(n) = (n+1) / 2 \in O(n)$$

当一般状况的时间复杂度存在时, $E(n) = B(n) = W(n) = A(n)$ 。

1.6 渐近式表示法

1.6.1 时间复杂度各类等级

上一节我们谈到时间复杂度的计算方式。如果现在有两个程序其目的相同，其时间复杂度分别为 $18n$ 和 n^2+3 ，到哪一个程序较有效率呢？答案是时间复杂度为 $18n$ 的程序较有效率，因为时间复杂度 $18n$ 的程序，当 n 愈大时，所需要的时间会远比时间复杂度 n^2+3 的程序短。我们可以从下列的表格中看出：

n	$18n$	n^2+3
1	18	4
10	180	103
100	1800	10003
1000	18000	1000003
10000	180000	100000003
100000	1800000	10000000003

在开始谈渐近式表示法以前，我们先了解一些时间复杂度的各种等级类型：

时间复杂度式为“ c ” ($c>0$)的算法称为常数时间算法(Constant-time algorithm)。

例如：1、3、7。

我们通常把时间复杂度式为“ $an+b$ ” ($a>0, b\geq 0$)的算法称为线性时间算法(Linear-time algorithm)。

例如： $3n+8$ 、 $7n+2$ 。

而时间复杂度式为“ an^2+bn+c ” ($a>0, b\geq 0, c\geq 0$)的算法称为平方时间算法(Quadratic-time algorithm)。在平方时间算法中，若再细分还可分为：

- 纯粹平方时间(Pure Quadratic-time)：

其时间复杂度式为“ an^2+c ”，其中 $a>0, c\geq 0$ 。例如： $6n^2$ 、 $7n^2+9$ 、 n^2 。

- 完全平方时间(Complete Quadratic-time)：

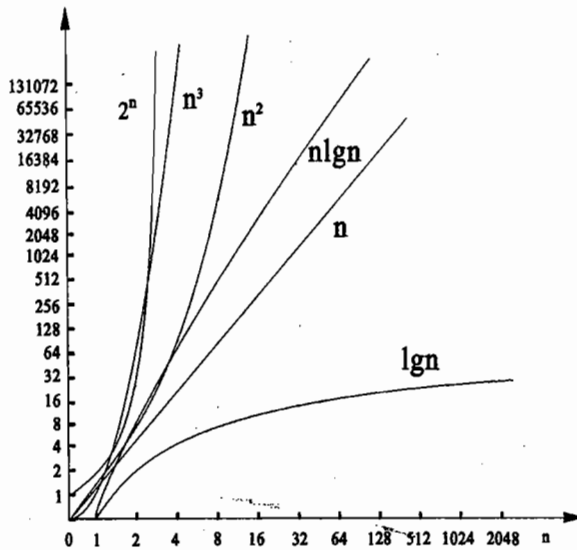
其时间复杂度式为“ an^2+bn+c ”，其中 $a>0, b>0, c\geq 0$ 。例如： $6n^2+n+4$ 、 $7n^2+n+9$ 、 n^2+n+6 。

时间复杂度式为“ an^3+bn^2+cn+d ” ($a>0, b\geq 0, c\geq 0, d\geq 0$)的算法称为立方时间算法(Cubic-time algorithm)。例如： n^3+5n^2+2n+8 。除此以外，“ $\log n$ ”为对数时间(Logarithmic-time)、“ a^n ”为指数时间(Exponential-time)。

其当 n 愈大时，其关系如下：

对数时间 < 线性时间 < 平方时间 < 立方时间 < 指数时间

即： $\log n < n < n^2 < n^3 < 2^n$ 。



1.6.2 渐近式表示法

渐近式表示法(Asymptotic Notations), 一般而言可分为下列 5 种:

1. $O()$ 表示法

其定义为: 对于 $f(n)$ 和 $g(n)$ 而言, 存在一些正实数常数 c 和非负整数 N , 使得在 $n \geq N$ 时, 满足

$$0 \leq f(n) \leq c \times g(n)$$

则称 $O(g(n)) = f(n)$ 。也就是 $g(n)$ 是 $f(n)$ 复杂度的渐近紧密上限(Asymptotic tight upper bound)。

例: 试证 $2n^2 + 10n \in O(n^2)$

证明:

此时 $f(n) = 2n^2 + 10n$, $g(n) = n^2$

由 $0 \leq f(n) \leq c \times g(n)$ 知

当 $n \geq 1$ 时, $0 \leq 2n^2 + 10n \leq 2n^2 + 10n^2 = 12n^2$ 恒成立。(c=12、N=1)

2. $\Omega()$ 表示法

其定义为: 对于 $f(n)$ 和 $g(n)$ 而言, 存在一些正实数常数 c 和非负整数 N , 使得在 $n \geq N$ 时, 满足

$$0 \leq c \times g(n) \leq f(n)$$

则称 $\Omega(g(n)) = f(n)$ 。也就是 $g(n)$ 为 $f(n)$ 复杂度的渐近紧密下限(Asymptotic tight lower bound)。

例: 试证 $2n^2 + 10n \in \Omega(n^2)$

证明:

此时 $f(n) = 2n^2 + 10n$, $g(n) = n^2$

由 $0 \leq c \times g(n) \leq f(n)$ 知

当 $n \geq 0$ 时, $2n^2 + 10n \geq n^2$ 恒成立。(c=1、N=0)

3. $\theta()$ 表示法

其定义为: 对于 $f(n)$ 和 $g(n)$ 而言, 存在一些正实数常数 c 、 d 和非负整数 N , 使得在 $n \geq N$ 时, 满足

$$0 \leq c \times g(n) \leq f(n) \leq d \times g(n)$$

则称 $\theta(g(n)) = f(n)$ 。也就是 $g(n)$ 为 $f(n)$ 复杂度的渐近紧密限度(Asymptotic tight bound)。 $\theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ 。

例: 试证 Exchange sort 算法 $\in \theta(n^2)$ 。

证明:

因为 $T(n) = n(n-1)/2 \in \Omega(n^2)$ 且 $\in O(n^2)$

所以由 $\theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ 知, 该算法 $\in \theta(n^2)$ 。

4. $o()$ 表示法

其定义为: 对于 $f(n)$ 和 $g(n)$ 而言, 对于任一正实数常数 c 和非负整数 N , 使得在 $n \geq N$ 时, 满足

$$0 \leq f(n) < c \times g(n)$$

则称 $o(g(n)) = f(n)$ 。也就是 $g(n)$ 为 $f(n)$ 复杂度的非紧密上限(Untightly upper bound)。

例: 试证 $n \notin o(5n)$

证明:

令 $c=1/6$, 如果 $n \in o(5n)$, 则必须存在一些 N , 使得对于 $n \geq N$ 时

$n \leq (1/6) * 5n = 5/6n$, 必须成立

得知该式不成立, 故矛盾, 所以 $n \notin o(5n)$ 。

5. $\omega()$ 表示法

其定义为: 对于 $f(n)$ 和 $g(n)$ 而言, 对于任一正实数常数 c 和非负整数 N , 使得在 $n \geq N$ 时, 满足于

$$0 \leq c \times g(n) < f(n)$$

则称 $\omega(g(n)) = f(n)$ 。也就是 $g(n)$ 为 $f(n)$ 复杂度的非紧密下限(Untightly lower bound)。

渐近式表示法虽然很多, 但判断时还是有一些规则可帮助我们很快的判断出其算法的效率, 为了方便读者判断, 笔者将规则整理成下列几点:

- 规则 1: 若 $b > 1$ 且 $a > 1$, 则

$$\log_a n \in \theta(\log_b n)$$

这一规则表示所有的对数时间皆属于同一类的复杂度, 所以我们将所有的对数时间复杂度皆归于 $\theta(\lg n)$ 这一类。

- 规则 2: 若 $b > a > 0$, 则

$$a^n \in o(b^n)$$

这一规则表示所有的指数时间皆属于不同类的复杂度。

- 规则 3: 若 $a > 0$, 则

$$\log n \in o(a^n)$$

这一规则表示指数的时间复杂度比较数的时间复杂度差。

- 规则 4: 若 $a > 0$, 则

$$a^n \in o(n!)$$

这一规则表示 $n!$ 的时间复杂度比指数的时间复杂度差。

- 规则 5: 若 $a \leq b$, 则

$$n^a \in O(n^b)$$

这一规则表示当 b 大于或等于 a 时, n^b 的时间复杂度则大于或等于 n^a 的时间复杂度。

- 规则 6: 若 $a > 0$ 且 $b > 1$, 则

$$n^a \in o(b^n)$$

这一规则表示 b^n 的时间复杂度则大于 n^a 的时间复杂度。

- 规则 7:

当 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ 时: 表示 $f(n) = g(n)$, 则 $f(n) \in \theta(g(n))$

当 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ 时: 表示 $f(n) < g(n)$, 则 $f(n) \in o(g(n))$

当 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ 时: 表示 $f(n) > g(n)$, 则 $g(n) \in o(f(n))$

1.7 递归式的复杂度计算

在前面的章节中, 我们谈了许多的时间复杂度的计算方式, 可是有时候, 一个算法的时间复杂度并不如预期的那么轻易地可以判别出来, 我们也许可以轻易地判别一个 for 循环的时间复杂度, 我们也许对于单一语句的时间复杂度也可以马上算出, 但是有时候遇到递归程序时, 我们却无法轻易地算出其时间复杂度。在这一节就是要让读者了解如何解递归算法中递归式的时间复杂度。

例如: 我们可以轻易地判断出一个算法执行指令数为 $n^3 + 2n^2 + 3$ 的时间复杂度, 但对于费氏级数 $F_n = F_{n-1} + F_{n-2}$ 却无从判断。如果我们将费氏级数解成一个等式的话, 那么对于判断时间复杂度, 就很方便了!

什么是“同质线性递归式(Homogeneous Linear Recurrences)”呢? 其定义为对于 k 和 a_i 皆为常数的“ $a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} + \dots + a_k t_{n-k} = 0$ ”数学算式, 称为同质线性递归式, 如: $t + 3t_{n-1} + 2t_{n-2} = 0$ 。

这类的递归式解法如下:

假设原式为 “ $a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} + \dots + a_k t_{n-k} = 0$ ”

步骤 1: 令 $t_n = r^n$, 则代换后其特征方程式(Characteristic Equation)为

$$a_0 r^n + a_1 r^{n-1} + a_2 r^{n-2} + \dots + a_k r^{n-k} = 0$$

步骤 2: 解特征方程式的解, 若有 k 个解为 $r_1, r_2, r_3, \dots, r_k$, 则

$$t_n = c_1 r_1^n + c_2 r_2^n + \dots + c_k r_k^n$$

步骤 3: 依题意所提供的条件代入解出 c_i 的值, 即可求出其递归等式。

例: 试求费氏级数的时间复杂度。

解:

已知费氏级数的递归式为 $F_n = F_{n-1} + F_{n-2}$ 且 $F_0 = 0, F_2 = 1$

(费氏级数: 0、1、1、2、3、5、8...)

令 $F_n = r^n$

则, 原式 $\Rightarrow r^n - r^{n-1} - r^{n-2} = 0$

同除以 r^{n-2}

则, 原式 $\Rightarrow r^2 - 1 - 1 = 0$

其两个根为 $\frac{1+\sqrt{5}}{2}$ 和 $\frac{1-\sqrt{5}}{2}$

所以, $F_n = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n + c_3 0^n$ (其中 $c_3 0^n$ 可略)

依题意知 $t_0=0, t_1=1$, 代入后得:

$$t_0 = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^0 + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^0 = 0 \quad (1)$$

$$t_1 = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^1 + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^1 = 1 \quad (2)$$

$$\text{解得 } c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}$$

$$\text{则: } t_n = \left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] / \sqrt{5}$$

从这个等式中我们可以轻易地看出, 费氏级数的时间复杂度属于指数时间。

例: 对于 $n>1$ 时, $t_n - 5t_{n-1} + 6t_{n-2} = 0$, 已知 $t_0=0, t_1=1$, 试求出递归等式。

解:

令 $t^n = r^n$

则, 原式 $\Rightarrow r^n - 5r^{n-1} + 6r^{n-2} = 0$

同除以 r^{n-2}

则, 原式 $\Rightarrow r^2 - 5r + 6 = 0, (r-2)(r-3) = 0$, 得 $r=2$ 或 3

所以, $t_n = c_1 3^n + c_2 2^n + c_3 0^n$ (其中 $c_3 0^n$ 可略)

依题意知 $t_0=0, t_1=1$, 代入后得:

$$t_0 = c_1 3^0 + c_2 2^0 = c_1 + c_2 = 0 \quad (1)$$

$$t_1 = c_1 3 + c_2 2 = 3c_1 + 2c_2 = 1 \quad (2)$$

解得 $c_1 = 1, c_2 = -1$

则, $t_n = 3^n - 2^n$

从这个等式中我们可以轻易地看出, 其时间复杂度属于指数时间。

如果解出来的根是重根的话, 则第1个根以 r^n 代入, 第2个重根以 nr^n 代入、第3个重根以 n^2r^n 代入、第4个重根以 n^3r^n 代入, 依此类推。

例: 对于 $n>2$ 时, $t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}$, 已知 $t_0=0, t_1=1, t_2=1$, 试求出递归等式。

解:

令 $t^n = r^n$

则, 原式 $\Rightarrow r^n - 5r^{n-1} + 8r^{n-2} - 4r^{n-3} = 0$

同除以 r^{n-3}

则, 原式 $\Rightarrow r^3 - 5r^2 + 8r - 4 = 0, (r-2)^2(r-1) = 0$, 得 $r=2$ (重根)或 1

所以, $t_n = c_1 2^n + c_2 n 2^n + c_3 1^n$

依题意知 $t_0=0, t_1=1, t_2=1$ 代入后得:

$$t_0 = c_1 2^0 + c_2 0 * 2^0 + c_3 1^0 = c_1 + c_3 = 0 \text{-----(1)}$$

$$t_1 = c_1 2 + c_2 1 * 2 + c_3 = 2c_1 + 2c_2 + c_3 = 1 \text{-----(2)}$$

$$t_2 = c_1 2^2 + c_2 2 * 2^2 + c_3 = 4c_1 + 8c_2 + c_3 = 1 \text{-----(2)}$$

解得 $c_1 = 3, c_2 = -1, c_3 = -3$

$$\text{则, } t_n = 3 * 2^n - n * 2^n - 3$$

从这个等式中我们可以轻易地看出其时间复杂度属于指数时间。

【习题】

一、复习：(是非、选择)

1. 算法是指为了完成某项特定工作所设计出的一连串用来说明工作是如何被完成的步骤。
2. 伪码是直接以程序语法来描述解决问题的方法。
3. 有一个程序片断如下：

```
for (I=0; I<n; I++)
    X=X+1;
```

则其时间复杂度为 $O(n)$ 。

4. 有一个程序片断如下：

```
for (I=0; I<n; I++)
    for (J=I; J<n; J++)
        for (K=J; K<n; K++)
            M = 1;
```

则时间复杂度为 $O(n^3)$ 。

5. 有一个程序片断如下：

```
X = 1000000
While ( X != 5 )
    X = X / 10
```

则时间复杂度为 $O(n)$ 。

6. 有一个程序片断如下：

```
for (I=0; I<n; I++)
    J = I
    While ( J >= 2 )
        J = J / 2
```

则时间复杂度为 $O(n^2)$ 。

7. 算法必须满足下列哪几点条件？
 - (a) 定义明确
 - (b) 有限的步骤
 - (c) 有效率的步骤
 - (d) 以上皆非
8. 下列哪项会影响程序中单一指令执行的时间(多选)？
 - (a) 机器的速度
 - (b) 循环范围
 - (c) 指令周期
 - (d) 编译器
9. 下列关于顺序查找法的语句哪一个错误？
 - (a) 最佳状况的时间复杂度为 $B(n) = 1 \in O(1)$ 。
 - (b) 最坏状况的时间复杂度为 $W(n) = n \in O(n)$ 。
 - (c) 一般状况的时间复杂度为 $E(n) = n \in O(n)$ 。
 - (d) 平均状况的时间复杂度为 $A(n) = (n+1)/2 \in O(n)$ 。
10. 有一个程序片断如下：


```
for (I=0; I<N; I++)
```

```

for (J=I+1; J<N; J++)
    if ( X[I] < X[J] )
    {
        Temp = X[I];
        X[J] = X[I];
        X[I] = Temp;
    }

```

当 $N=6$, 且 $X[0]=5$, $X[1]=9$, $X[2]=2$, $X[3]=6$, $X[4]=7$, $X[5]=4$, 运行完这个程序片断后, 则下列哪一个语句错误?

- (a) $X[2]=6$ (b) $X[5]=2$ (c) $X[3]=5$ (d) $X[1]=9$
11. 从 n 个未排列的数中, 找出中位数的计算复杂度为哪一个?
 (a) $O(1)$ (b) $O(n)$ (c) $O(n \log n)$ (d) $O(n^2)$
12. 下列语句哪一个正确?
 (a) $n! \in O(a^n)$, a 为正整数 (b) $n^{0.00001} \in O(\log n)$
 (c) $1 + 1/2 + 1/3 + 1/4 + \dots + 1/n \in O(\log n)$ (d) $3^n \in O(2^n)$
 (e) 以上皆非。

二、应用:

1. 如何判断一个算法的好坏?
2. 假设程序中每一个指令运行的时间为 t , 影响 t 值的因素有哪些?
3. 试求二分查找法的 $B(n)$ 、 $W(n)$ 、 $A(n)$ 。
4. 试由大到小排列出以下函数的时间复杂度。

$$F1(n) = n^2 \log n + \log n$$

$$F2(n) = n/100 + 100/n^2$$

$$F3(n) = 2 \log(\log n)$$

$$F4(n) = (\log n)^2$$

$$F5(n) = 2 \log n$$

$$F6(n) = \log n^2$$

$$F7(n) = \log(n + 2/n)$$

5. $n^2 \log n \in O(n^2)$, 这个语句成立或不成立? 试证明你的答案。
6. 试求对于 $n > 1$, $t_n = 6t_{n-1} - 9t_{n-2}$ 的时间复杂度。已知 $t_0=0$, $t_1=1$ 。

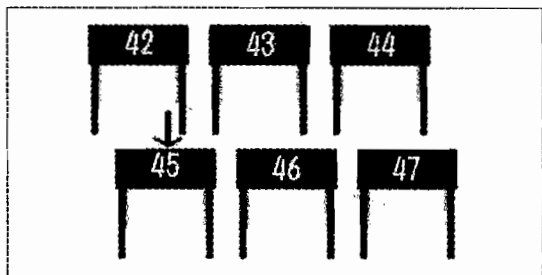
数 组

第 2 章

- ◆ 何谓数组
- ◆ 一维数组
- ◆ 一维数组的使用
- ◆ 一维数组的存取
- ◆ 一维数组的遍历
- ◆ 一维数组的高级应用
- ◆ 二维数组
- ◆ 数组表示法
- ◆ 特殊类型的数组

2.1 何谓数组

数组(Array)是数据结构中最基本的结构类型。我们可以运用生活中的例子来描述一下计算机中数组的样子,数组像是学校中每个学生都有一个属于自己的抽屉,假设抽屉的编号是以学生的座号来命名,由此得知,当我们想把某样东西交给 45 号同学时,我们只要把该东西放入 45 号的抽屉里,45 号同学就可以在 45 号抽屉中拿到该东西,这种用循序编号抽屉来存取数据的方式,也就是数组存取数据的方式。



数组是一种循序式的结构,假设定义出数组大小为 0 到 100,则需预留 101 个存储空间,缺一不可(但前段所述的抽屉概念,却可能因为这个座号没有人而无需事先预留抽屉)。对于每一个数组而言,都有一个索引值(Index)和一个内容值(Value),索引值是可使数据方便存取(正如前述抽屉概念中的座号),而内容值正是该笔被存储的数据。

数组是存储同一类型数据的数据结构。数组使用的是一种静态的内存空间配置,静态内存就是程序设计者必需在程序设计时,就要把所需的内存空间大小和数据类型给定义出,程序在编译的过程中便会依程序设计者的定义将空间给配置出来,静态内存虽然可将程序所需的空间和数据使用的数据类型事先定义出,但却缺乏使用弹性,因为程序设计者在设计程序时,并无法去预算所需的空间和数据使用的数据类型,如果程序设计者定义出的数组空间过小,那么在使用时易造成程序的执行错误,反之如果程序设计者定义出的数组空间过大,那么造成内存空间的浪费。

如果需要让内存的使用弹性更高,则需使用动态的内存配置,动态的记忆配置方式,我们将在以后链表那一章节中有更详尽的介绍。在这一章中,我们将学习一维数组、二维数组、甚至多维数组的使用方法。

2.2 一维数组

一维数组是数组结构中最基本的类型,一维数组正如同前述的抽屉概念一样,是由编号(索引值)和空间(内容值)来定义。在 C 语言中,使用数组前必需事先声明,程序才能在编译的过程中预留内存空间。

- 声明格式:

数据类型 变量名称[长度];

- 程序范例:

```
int    NumberA[10];
//声明一个长度为 10 的整数数据类型一维数组,数组名为 NumberA
```

● 程序说明:

上述的“`int NumberA[10];`”声明,可以用下列图形来说明,帮助读者更直观地了解数组在内存中配置。

因为声明为 `int`(整数数据类型),而每一个整数数据类型所占的内存空间为 16 位,共 2 个字节,假设数组的第一个元素的内存位置为 `X`。

$\text{NumberA}[i]$ 的内存位置 = 数组第一个元素位置 + ($i * \text{所声明数据类型所占的大小}$)

所以我们画出下面这个图形:

内存位置		数组名
<code>X</code>		<code>NumberA[0]</code>
<code>X+(1*2)</code>		<code>NumberA[1]</code>
<code>X+(2*2)</code>		<code>NumberA[2]</code>
<code>X+(3*2)</code>		<code>NumberA[3]</code>
<code>X+(4*2)</code>		<code>NumberA[4]</code>
<code>X+(5*2)</code>		<code>NumberA[5]</code>
<code>X+(6*2)</code>		<code>NumberA[6]</code>
<code>X+(7*2)</code>		<code>NumberA[7]</code>
<code>X+(8*2)</code>		<code>NumberA[8]</code>
<code>X+(9*2)</code>		<code>NumberA[9]</code>

● 程序范例:

```
float NumberB[5];
//声明一个长度为 5 的浮点数据类型一维数组,数组名为 NumberB
```

● 程序说明:

同理对于上述的“`float NumberB[5];`”声明,用以下图形来说明,帮助读者更直观地了解数组在内存中配置。

因为声明为 `float`(浮点数数据类型),而每一个浮点数数据类型所占的内存空间为 32 位(bits),共 4 个字节(byte),假设数组的第一个元素的内存位置为 `X`。

$\text{NumberB}[i]$ 的内存位置 = 数组第一个元素位置 + ($i * \text{所声明数据类型所占的大小}$)

所以我们画出下面这个图形:

内存位置		数组名
<code>X</code>		<code>NumberB[0]</code>
<code>X+(1*4)</code>		<code>NumberB[1]</code>
<code>X+(2*4)</code>		<code>NumberB[2]</code>
<code>X+(3*4)</code>		<code>NumberB[3]</code>
<code>X+(4*4)</code>		<code>NumberB[4]</code>

● 补充说明:

在 C 语言中,数组声明可使用的数据类型除了整数、浮点数,还可声明成字符串数据类型,甚至可以声明为结构数组。

结构的声明如下:

```
struct PhoneBook
{
    string    Name;
```

```

        int      Telephone;
        int      MobilePhone;
    }

```

结构数组的声明如下：

```
struct PhoneBook Data[7];
```

假设该结构所占的内存空间为 m ，我们可用 `sizeof(struct PhoneBook)` 来算出该结构所占的空间。该结构数组的图形如下：

内存位置		数组名
X		Data[0]
$X+(1*m)$		Data[1]
$X+(2*m)$		Data[2]
$X+(3*m)$		Data[3]
$X+(4*m)$		Data[4]
$X+(5*m)$		Data[5]
$X+(6*m)$		Data[6]

在 C 语言中，数组声明的长度，索引值都从 0 开始计数，有些程序语言则是从 1 开始计数，而像 Pascal 则可依用户需要设置数组的索引值的起始值和终止值。下面我举一个 Pascal 的数组声明格式供读者参考。

Pascal 数组声明格式：

```

type
    数组类型 = array[数组范围] of 数据类型

```

Pascal 数组程序范例：

```

type
    RealArray = array[3..9] of real;

var
    NumberA : RealArray;

```

Pascal 数组程序说明：

以上的程序是定义一个 `RealArray` 的数组类型，其数组范围是从 3 到 9(3、4、5、6、7、8、9)，共 7 个空间，其数组数据类型为实数数据类型。而我们声明一个 `NumberA` 变量为 `RealArray` 数据类型，表示 `NumberA` 为一个具有从 3 到 9 的实数数组。

2.3 一维数组的使用

一维数组的运用范围很广，在此我们先以一个简单的例子来说明，让读者对数组有些基本的认识。

程序实例：

运用一维数组设计一个可计算多个数的平均值的程序。

程序构思：

依题意可得知，本程序所运用的数据结构为“一维数组”，目的为“可计算多个数的平均值”。

数组是本程序中用于存储数据的空间，我们先默认一个大小为 20 的浮点数数组，并提供用户程序执行之初输入欲输入数据的个数(小于 20)。

程序执行时先将数据加总变量初值设置为 0。并要求用户输入欲输入数据的个数。

经过 if 的判断句来判断用户输入是否超过 20，当用户输入小于 20 时，执行数据输入循环内的程序，否则执行输出 "Please input a number less than 20." 消息。

循环中的程序功用为读取用户输入的数据，并累加至 Summary 变量中。

平均值的求法为数据总和除以数据个数。

程序源代码：

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-01.c */
03  /* 程序目的: 运用一维数组设计一个可计算多个数的平均值 */
04  /* 的程序。 */
05  /* Written By Kuo-Yu Huang. (WANT Studio.) */
06  /* ===== */
07  void main ( )
08  {
09      int    Max=0;          /* 最大可输入数据个数 */
10      int    I;             /* for 循环计数变量 */
11      float  Number[20];    /* 存储数据的浮点数数组 */
12      float  Summary;       /* 数据加总变量 */
13      float  Average;       /* 数据平均变量 */
14
15      Summary = 0;          /* 数据加总变量初始化 */
16      printf("Please input the number of data:");
17      scanf("%d",&Max);
18      if (Max <= 20)        /* 决定最大可输入数 */
19      {
20          for (i=0;i<Max;i++) /* 数据输入循环 */
21          {
22              printf("Please input a number:");
23              scanf("%f",&Number[i]);
24              Summary += Number[i];
25          }
26          Average = Summary / Max; /* 平均值 = 总和 / 个数 */
27          printf("The average is %5.2f \n",Average);
28      }
29      else
30          printf("Please input a number less than 20.");
31  }

```

说明

上述程序中第 01~06 行为程序批注，本书中程序的行号为方便读者阅读，实际的 C 语言编写并无行号。本书中在每个程序开头都有程序批注说明这程序的文件名和程序目的。

运行结果：

```

C:\DS>array-01
Please input the number of data:5
Please input a number:13.5
Please input a number:170.4
Please input a number:65.45
Please input a number:84
Please input a number:20
The average is 70.67

```

```

C:\DS>

```

2.4 一维数组的存取

在数组中,数据是可以随机存取的,只要我们给个索引值,就可马上找到该索引值所对应数组的数据,也就是说利用索引值,即可找到该数组的内容值。比如:我们现在利用一个数组来存储员工的工资,其内容如下:

0	1	2	3	4	5	6	7	8	9
27000	32000	32500	27500	28500	29000	31000	32500	30000	26000

索引值所代表的正是员工的编号,而内容值所代表的是员工的工资。当我们今天想查编号为 5 号的员工工资数据,我们马上可由上图对应出编号为 5 号的员工工资数据是 29000。或者当编号为 4 号员工的工资需要调整时,我们也可利用索引值,将修改后的数据存回。

程序实例:

运用一维数组设计一个简易的员工工资管理系统(具有查询和修改功能)。

程序构思:

依题意可得知,本程序所运用的数据结构为“一维数组”,目的为“具有查询和修改功能的员工工资管理系统”。

我们先默认一个大小为 10 的整数数组,并预先依员工编号设置出 10 笔员工工资数据。

利用一个 while 循环提供一个用户菜单供用户选择功能,第一项功能为查询员工数据、第二项功能为修改员工数据、第 3 项为结束系统。

当用户选择为 1 或 2 时,先要求用户输入员工编号,程序再进行判断用户输入是否为正确的员工编号,正确时,则输出该员工的工资数据,错误时,则输出“## The error employee number!!”。

若用户选择为 2 时,则再进一步要求用户输入修改后新的员工工资数据,并将数据存回该员工工资数据数组中。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-02.c */
03  /* 程序目的: 运用一维数组设计一个简易的员工工资管理系 */
04  /*          统(具查询和修改功能)。 */
05  /* Written By Kuo-Yu Huang. (WANT Studio.) */
06  /* ===== */
07  void main ()
08  {
09          /* 默认 10 笔员工工资数据 */
10          int    Employee[10] = { 27000, 32000, 32500, 27500,
11                                28500, 29000, 31000, 32500,
12                                30000, 26000 };
13          int     Index;          /* 数组索引变量 */
14          int     NewSalary;      /* 修改后工资变量 */
15          int     Selection;      /* 用户选项变量 */
16
17          while ( 1 )             /* while 循环结构 */
18          {
19
20                      /* 用户菜单 */
21          printf("===== \n");
22          printf("= Simple Employee Salary Management System = \n");

```

```

22     printf("= 1.Display employee salary           =\n");
23     printf("= 2.Modify employee salary           =\n");
24     printf("= 3.Quit                               =\n");
25     printf("===== \n");
26     printf("Please input your choose:");
27     scanf("%d",&Selection); /* 读取用户输入选项 */
28
29     if (Selection == 1 || Selection == 2)
30     {
31         printf("*** Please input the employee number:");
32         scanf("%d",&Index); /* 读取员工编号 */
33         if (Index < 10)      /* 判断员工编号的范围 */
34         {
35             printf("*** Employee Number is %d .",Index);
36             printf("The Salary is %d\n",Employee[Index]);
37         }
38         else
39         {
40             printf("## The error employee number!!\n");
41             exit(1);
42         }
43     }
44
45     switch (Selection)
46     {
47         case 1: /* 选项 1:输出员工工资 */
48             break; /* 选项 2:修改员工工资 */
49         case 2:
50             printf("*** Please input new salary:");
51             scanf("%d",&NewSalary);/* 读取修改后员工工资 */
52             Employee[Index] = NewSalary;
53             break;
54         case 3: /* 选项 3:结束程序 */
55             exit(1);
56             break;
57     }
58     printf("\n");
59 }
60 }

```

运行结果:

```

C:\DS>array-02
=====
= Simple Employee Salary Management System =
= 1.Display employee salary                 =
= 2.Modify employee salary                 =
= 3.Quit                                   =
=====
Please input your choose:1
** Please input the employee number:5.
** Employee Number is 5 .The Salary is 29000

=====
= Simple Employee Salary Management System =
= 1.Display employee salary                 =
= 2.Modify employee salary                 =
= 3.Quit                                   =
=====
Please input your choose:2
** Please input the employee number:5

```

```

** Employee Number is 5 .The Salary is 29000
** Please input new salary:30000

=====
= Simple Employee Salary Management System =
= 1.Display employee salary                  =
= 2.Modify employee salary                  =
= 3.Quit                                    =
=====
Please input your choose:1
** Please input the employee number:5
** Employee Number is 5 .The Salary is 30000

=====
= Simple Employee Salary Management System =
= 1.Display employee salary                  =
= 2.Modify employee salary                  =
= 3.Quit                                    =
=====
Please input your choose:3

C:\DS>

```

2.5 一维数组的遍历

有时候,如果想知道一个数值是否存在于数组中,我们就无法利用索引值去找寻数据,在这个时候,我们所运用的就是将数值与数组中的每一个内容值做比较,循序地改变索引值,并比较每一个数组的内容值,这就是数组的遍历(Traverse)。

程序实例:

运用一维数组设计一个可存储员工工资,让用户输入工资,然后输出员工编号的程序。

程序构思:

依题意可得知,本程序所运用的数据结构为“一维数组”,目的为运用遍历数组的方式“查询员工工资数据”。

首先默认一个大小为 10 的整数数组,并预先依员工编号设置出 10 笔员工工资数据。

在 for 循环中判断当用户输入的工资等于数组中的内容值时,则输出数据,将计数器加一,并依序将每个数组中的数据都遍历比较一次。

程序最后再判断数据计数是否为零,当数据计数为零时,表示找不到相符的工资数据,若不为零,则输出共有几笔相符的数据。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-03.c */
03  /* 程序目的: 运用一维数组设计一个可存储员工工资,让使 */
04  /* 用者输入工资,然后输出员工编号的程序。 */
05  /* Written By Kuo-Yu Huang. (WANT Studio.) */
06  /* ===== */
07  void main ()
08  {
09      /* 默认 10 笔员工工资数据 */

```

```

10     int    Employee[10] = { 27000, 32000, 32500, 27500,
11                               28500, 29000, 31000, 32500,
12                               30000, 26000 };
13     int Salary; /* 用户输入工资变量 */
14     int Counter=0; /* 数据计数变量 */
15     int i; /* for 循环计数变量 */
16
17     printf("Please input the employee salary: ");
18     scanf("%d",&Salary); /* 读取欲查询工资数据 */
19     for (i=0;i<10;i++) /* 数组遍历循环 */
20         if (Salary == Employee[i]) /* 当数据找到时 */
21         {
22             printf("Number %d Employee's salary is %d !\n",i,Salary);
23             Counter++; /* 数据计数变量加一 */
24         }
25     if (Counter == 0) /* 当数据计数为零时 */
26         printf("No employee's salary is %d!!\n",Salary);
27     else /* 当数据计数不为零时 */
28         printf("Have%d employees salary is %d !!\n",Counter,Salary);
29 }

```

运行结果:

```

C:\DS>array-03
Please input the employee salary: 18500
No employee's salary is 18500!!
C:\DS>array-03
Please input the employee salary: 27000
Number 0 Employee's salary is 27000 !
Have 1 employees salary is 27000 !!
C:\DS>array-03
Please input the employee salary: 32500
Number 2 Employee's salary is 32500 !
Number 7 Employee's salary is 32500 !
Have 2 employees salary is 32500 !!
C:\DS>

```

2.6 一维数组的高级应用

一维数组除了以上基本的使用外,还可运用一维数组来存储数据,弥补 C 语言一些数据类型所无法达到的目的,比如,我们需要存储一个很大的数字,但使用 int 的整数数据类型范围却只能在-32768~32768 之间,按照要求,我们开始构思数组存储数据的方式,我们可运用数组来创造出可存储更大数字的空间。以下将以一个实例来说明:

程序实例:

设计一个可容纳 40 位数的求 n! 的程序。

程序构思:

依题意得知,本程序所运用的数据结构为“数组”,目的为运用数组来弥补整数数据类型有限的使用范围。

我们预先声明变量为一个大小为 40 的数组,负责存储每一个位数的数据,变量 Digit 为计算位数的变量、变量 i, j, r, k 为循环中所用的计数变量。

首先先将 Data 数组中的数据做初始值零。再令第一位数值为 1,位数也为 1。

再将每次相乘的乘积存回数组中。并循序处理每个数组中超过 10 的数，若数值超过 10，则将位数加一，原来的数除以 10，商数加前一位数的数值后存回前一位数的数组中，再将余数存回原来位数的数组中。最后再输出每次计算后的结果。

例如：我们想求 7!，则流程为：

步骤 1:

$$1! = 1$$

位数为 1

数组内容:

4	3	2	1
0	0	0	1

步骤 2:

$$2! = 2 * 1! = 2$$

位数为 1

$$\text{数组}[1] = \text{数组}[1] * 2$$

数组内容:

4	3	2	1
0	0	0	2

步骤 3:

$$3! = 3 * 2! = 3 * 2 = 6$$

位数为 1

$$\text{数组}[1] = \text{数组}[1] * 3$$

数组内容:

4	3	2	1
0	0	0	6

步骤 4:

$$4! = 4 * 3! = 4 * 6 = 24$$

位数为 1

$$\text{数组}[1] = \text{数组}[1] * 4$$

数组内容:

4	3	2	1
0	0	0	24

因为数组[1]的值大于 10 所以必须进位。

$$\text{数组}[2] = \text{数组}[2] + \text{数组}[1]/10 = 0 + 2 = 2$$

$$\text{数组}[1] = \text{数组}[1] \% 10 = 4$$

位数加 1(位数为 2)

所以数组内容为:

4	3	2	1
0	0	2	4

步骤 5:

$$5! = 5 * 4! = 5 * 24 = 120$$

位数为 2

$$\text{数组}[1] = \text{数组}[1] * 5$$

$$\text{数组}[2] = \text{数组}[2] * 5$$

数组内容:

4	3	2	1
0	0	2*5=10	4*5=20

因为数组[1]的值大于 10 所以必须进位。

$$\text{数组}[2] = \text{数组}[2] + \text{数组}[1]/10 = 10 + 2 = 12$$

$$\text{数组}[1] = \text{数组}[1] \% 10 = 0$$

因为数组[2]的值大于 10 所以必须进位。

$$\text{数组}[3] = \text{数组}[3] + \text{数组}[2]/10 = 0 + 1 = 1$$

$$\text{数组}[2] = \text{数组}[2] \% 10 = 2$$

位数加 1(位数为 3)

所以数组内容为:

4	3	2	1
0	1	2	0

步骤 6:

$$6! = 6 * 5! = 6 * 120 = 720$$

位数为 3

$$\text{数组}[1] = \text{数组}[1] * 5$$

$$\text{数组}[2] = \text{数组}[2] * 5$$

$$\text{数组}[3] = \text{数组}[3] * 5$$

数组内容:

4	3	2	1
0	1*6=6	2*6=12	0*6=0

因为数组[2]的值大于 10, 所以必须进位。

$$\text{数组}[3] = \text{数组}[3] + \text{数组}[2]/10 = 6 + 1 = 7$$

$$\text{数组}[2] = \text{数组}[2] \% 10 = 2$$

所以数组内容为:

4	3	2	1
0	7	2	0

步骤 7:

$$7! = 7 * 6! = 7 * 720 = 5040$$

位数为 3

$$\text{数组}[1] = \text{数组}[1] * 7$$

$$\text{数组}[2] = \text{数组}[2] * 7$$

$$\text{数组}[3] = \text{数组}[3] * 7$$

数组内容:

4	3	2	1
0	7*7=49	2*7=14	0*7=0

因为数组[2]的值大于 10, 所以必须进位。

$$\text{数组}[3] = \text{数组}[3] + \text{数组}[2]/10 = 49 + 1 = 50$$

$$\text{数组}[2] = \text{数组}[2] \% 10 = 4$$

因为数组[3]的值大于 10, 所以必须进位。

$$\text{数组}[4] = \text{数组}[4] + \text{数组}[3]/10 = 0 + 5 = 5$$

$$\text{数组}[3] = \text{数组}[3] \% 10 = 0$$

所以数组内容为:

4	3	2	1
5	0	4	0

步骤 8:

输出数组数据为 5040。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-04.c */
03  /* 程序目的: 设计一个可容纳 40 位数的求 n! 程序 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  void main(void)
07  {
08      int    Data[40];      /* 存储 40 位数的整数数组 */
09      int    Digit;        /* 数据位数变量 */
10      int    i,j,r,k;      /* 循环计数变量 */
11      int    N;            /* 用户输入值 */
12
13      for (i=1;i<40+1;i++) /* 将数组初始值设为 0 */
14          Data[i]=0;
15
16      Data[0]=1;           /* 设第 0 位数数组为 1 */
17      Data[1]=1;           /* 设第 1 位数数组为 1 */
18      Digit=1;            /* 设数据位数为 1 */
19
20      printf("Enter a number what you want to calculus : ");
21      scanf("%d",&N);      /* 读取用户欲求的 N 值 */
22
23      for (i=1;i<N+1;i++)
24      {
25          for (j=1;j<Digit+1;j++)
26              Data[j]*=i;    /* 数组中内容的运算 */
27          for (j=1;j<Digit+1;j++)
28          {
29              if (Data[j]>10)
30              {
31                  for (r=1;r<Digit+1;r++)
32                  {
33                      if (Data[Digit]>10)
34                          Digit++;
35                      /* 当数组中的值大于 10 时, 则位数加 1 */
36                      Data[r+1] += Data[r]/10;
37                      /* 前一位数组值 = 前一位数组值 + 此位数组值除以 10 */
38
39                      Data[r]=Data[r]%10;
40                      /* 此位数组值 = 此位数组值除 10 取余数 */
41                  }
42              }
43          }
44          printf("%d! = ",i);
45          for (k=Digit;k>0;k--) /* 输出数组中的内容 */
46              printf("%d",Data[k]);
47          printf("\n");
48      }
49  }

```

运行结果:

```

C:\DS>array-04
Input a number what you want to calculus :34
1! = 1
2! = 2

```

```

3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 51090942171709440000
22! = 112400072777607680000
23! = 25852016738884976640000
24! = 620448401733239439360000
25! = 15511210043330985984000000
26! = 403291461126605635584000000
27! = 0888869450418352160768000000
28! = 304888344611713860501504000000
29! = 8841761993739701954543616000000
30! = 265252859812191058636308480000000
31! = 8222838654177922817725562880000000
32! = 263130836933693530167218012160000000
33! = 8683317618811886495518194401280000000
34! = 295232799039604140847618609643520000000
C:\DS>

```

39.

2.7 二维数组

除了一维数组外，C 语言还提供了二维数组，所谓的二维数组，其实我们可以把它想象成是一个二维空间，在一维数组中的每一个数据都是在一行，所以我们只要指定我们要第几个数据，就可找到该笔数据的内容，而二维数组中的每一个数据都是由行和列所构成的，比如说，我们现在有一个置物柜，置物柜上共有 5 层，而每一层中有 4 格个小格子，我们就不能像一维数组那样，只指定第几个数据，而需要指定是第几列的第几个小格子，这就是二维数组的概念。

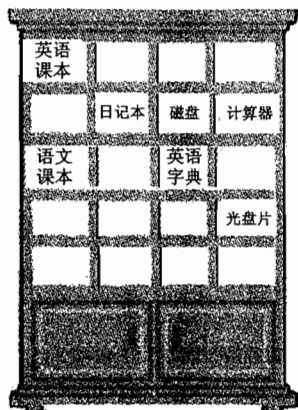
下图可帮读者了解二维数组，假设有一个置物柜如下图：

我们该如何描述日记本在哪呢？我们一定会说“日记本在置物柜上的第 2 层第 2 个格子中。”（由上往下，由左往右数）

没错！二维数组的描述方式也是如此，比如我们声明一个二维数组如下：

```
int Data[5][4];
```

C 语言便会在内存中为我们配置出一个大小为 5*4 的二维空间，如下表：



	第 1 列	第 2 列	第 3 列	第 4 列
第 1 行	(0,0)	(0,1)	(0,2)	(0,3)
第 2 行	(1,0)	(1,1)	(1,2)	(1,3)
第 3 行	(2,0)	(2,1)	(2,2)	(2,3)
第 4 行	(3,0)	(3,1)	(3,2)	(3,3)
第 5 行	(4,0)	(4,1)	(4,2)	(4,3)

因为声明为 `int`(整数数据类型), 而每一个整数数据类型所占的内存空间为 16 位, 共 2 个字节, 假设数组的第一个元素的内存位置为 X 。

$\text{Data}[i][j]$ 的内存位置

$$= \text{数组第一个元素位置} + [(i * \text{每一列的元素个数}) + j] \\ * (\text{所声明数据类型所占的大小})$$

所以我们可以画出下面这个图形:

内存位置		数组名
$X + (0 * 4 + 0) * 2$ $= X$		$\text{Data}[0][0]$
$X + (0 * 4 + 1) * 2$ $= X + 2$		$\text{Data}[0][1]$
$X + (0 * 4 + 2) * 2$ $= X + 4$		$\text{Data}[0][2]$
$X + (0 * 4 + 3) * 2$ $= X + 6$		$\text{Data}[0][3]$
$X + (1 * 4 + 0) * 2$ $= X + 8$		$\text{Data}[1][0]$
$X + (1 * 4 + 1) * 2$ $= X + 10$		$\text{Data}[1][1]$
\vdots	\vdots	\vdots
$X + (4 * 4 + 0) * 2$ $= X + 32$		$\text{Data}[4][0]$
$X + (4 * 4 + 1) * 2$ $= X + 34$		$\text{Data}[4][1]$

$$X+(4*4+2)*2$$

$$=X+36$$

$$X+(4*4+3)*2$$

$$=X+38$$

Data[4][2]

Data[4][3]

程序实例:

设计一个矩阵的相乘的程序,

假设

$$A = \begin{bmatrix} 1 & 5 & 7 & 3 \\ 3 & 6 & 3 & 9 \\ 1 & 2 & 8 & 7 \\ 0 & 3 & 1 & 9 \\ 3 & 2 & 5 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 9 & 1 & 4 & 1 & 4 \\ 5 & 6 & 7 & 9 & 0 & 3 \\ 3 & 2 & 7 & 2 & 5 & 6 \\ 9 & 7 & 4 & 7 & 8 & 0 \end{bmatrix}$$

求出 A*B 矩阵。

程序构思:

依题意可知, 本程序所运用的数据结构为 “二维数组”, 目的为运用数组来存储矩阵数据, 并进行矩阵乘法运算。

我们所知的矩阵乘法运算的算式如下:

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

故可运用一个 3 层的循环来运算此算式。

由题意知流程如下:

则

$$\begin{aligned} C(1,1) &= A(1,1) * B(1,1) + A(1,2) * B(2,1) + A(1,3) * B(3,1) + A(1,4) * B(4,1) \\ &= (1 * 3) + (5 * 5) + (7 * 3) + (3 * 9) \\ &= 3 + 25 + 21 + 27 \\ &= 76 \end{aligned}$$

同理

$$\begin{aligned} C(1,2) &= A(1,1) * B(1,2) + A(1,2) * B(2,2) + A(1,3) * B(3,2) + A(1,4) * B(4,2) \\ &= (1 * 9) + (5 * 6) + (7 * 2) + (3 * 7) \\ &= 9 + 30 + 14 + 21 \\ &= 74 \end{aligned}$$

依此类推, 我们可以求得矩阵 A 与矩阵 B 的矩阵乘积。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-05.c */
03  /* 程序目的: 设计一个矩阵的相乘程序 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  void main(void)
07  {
08      /* 默认 5*4 的矩阵数据 */
09      int MatrixA[5][4] = { 1, 5, 7, 3,
10                          3, 6, 3, 9,
11                          1, 2, 8, 7,
12                          0, 3, 1, 9,
13                          3, 2, 5, 4 };
14      /* 默认 4*6 的矩阵数据 */
15      int MatrixB[4][6] = { 3, 9, 1, 4, 1, 4,
16                          5, 6, 7, 9, 0, 3,
17                          3, 2, 7, 2, 5, 6,
18                          9, 7, 4, 7, 8, 0 };
19      int MatrixC[5][6]; /* 声明 5*6 的矩阵 */
20      int i, j, k; /* 循环计数变量 */
21
22      for (i=0; i<5; i++)
23          for (j=0; j<6; j++)
24          {
25              MatrixC[i][j] = 0; /* 设置 MatrixC 数组初始值为 0 */
26              for (k=0; k<4; k++)
27                  /* 进行矩阵乘法 C(i,j) = C(i,j) + A(i,k) * B(k,j) */
28                  MatrixC[i][j] += MatrixA[i][k] * MatrixB[k][j];
29          }
30      /* 输出 MatrixA 的数据 */
31      printf("The Matrix A:\n");
32      for (i=0; i<5; i++)
33      {
34          for (k=0; k<4; k++)
35              printf("%5d", MatrixA[i][k]);
36          printf("\n");
37      }
38      /* 输出 MatrixB 的数据 */
39      printf("\nThe Matrix B:\n");
40      for (k=0; k<4; k++)
41      {
42          for (j=0; j<6; j++)
43              printf("%5d", MatrixB[k][j]);
44          printf("\n");
45      }
46      /* 输出 MatrixC 的数据 */
47      printf("\nMatrix C = Matrix A * Matrix B\n");
48      for (i=0; i<5; i++)
49      {
50          for (j=0; j<6; j++)
51              printf("%5d", MatrixC[i][j]);
52          printf("\n");
53      }
54  }

```

运行结果:

```
C:\DS>array-05
The Matrix A:
  1   5   7   3
  3   6   3   9
  1   2   8   7
  0   3   1   9
  3   2   5   4

The Matrix B:
  3   9   1   4   1   4
  5   6   7   9   0   3
  3   2   7   2   5   6
  9   7   4   7   8   0

Matrix C = Matrix A * Matrix B :
  76  74  97  84  60  61
 129 132 102 135  90  48
 100  86  99  87  97  58
  99  83  64  92  77  15
  70  77  68  68  60  48

C:\DS>
```

2.8 数组表示法

介绍完了一维数组和二维数组之后,读者一定正在猜想除了一维数组、二维数组外,是不是还有三维数组、四维数组,甚至是10维数组呢?二维数组在内存中的存储方式是不是只能依着“循序存储每列的数据”这种方式而已呢?

不错,数组除了一维和二维外,还可以依需要定义出多维数组。而对于二维数组的存储方式并不只有依“循序存储每列的数据”这种方式而已,我们通常称这种数组的存储方式为“以行为主”(Row-Major),除此之外还有“以列为主”(Column-Major)的数组存储方式。

我们都清楚内存是一段循序的存储空间;并不是一个多维度的存储空间,而是一维度的存储空间,所以数组中的数据要存储于内存中,是依序的存储于此一维空间中,这也是C语言处理多维数组的方法。这一节我们所要谈的正是数组的表示法,让读者明白如何将一个多维数组转换成一维数组来表示。并藉此了解数组在内存中存储方式。

一般而言,数组的表示法可分为:以行为主和以列为主两种。以行为主的表示法规则为每一行排完后再排下一行,依序将每一行排入空间中;以列为主的表示法规则为每一列排完后再排下一列,依序将每一列排入空间中。

假设我们现在声明一个二维数组如下:

```
int Data[5][4];
```

如下表:

	第1列	第2列	第3列	第4列
第1行	(0,0)	(0,1)	(0,2)	(0,3)
第2行	(1,0)	(1,1)	(1,2)	(1,3)
第3行	(2,0)	(2,1)	(2,2)	(2,3)
第4行	(3,0)	(3,1)	(3,2)	(3,3)

第 5 行	(4,0)	(4,1)	(4,2)	(4,3)
-------	-------	-------	-------	-------

因为声明为 `int`(整数数据类型), 而每一个整数数据类型所占的内存空间为 16 位, 共 2 个字节, 假设数组的第一个元素的内存位置为 X 。

若以行为主来表示:

$\text{Data}[i][j]$ 的内存位置

$$= \text{数组第一个元素位置} + [(i * \text{每一行的元素个数}) + j] \\ * (\text{所声明数据类型所占的大小})$$

所以我们可以画出下面这个表:

	内存位置	数组名	行数
0	$X + (0 * 4 + 0) * 2$ $= X$	<code>Data[0][0]</code>	第 1 行
1	$X + (0 * 4 + 1) * 2$ $= X + 2$	<code>Data[0][1]</code>	
2	$X + (0 * 4 + 2) * 2$ $= X + 4$	<code>Data[0][2]</code>	
3	$X + (0 * 4 + 3) * 2$ $= X + 6$	<code>Data[0][3]</code>	
4	$X + (1 * 4 + 0) * 2$ $= X + 8$	<code>Data[1][0]</code>	第 2 行
5	$X + (1 * 4 + 1) * 2$ $= X + 10$	<code>Data[1][1]</code>	
⋮	⋮	⋮	⋮
16	$X + (4 * 4 + 0) * 2$ $= X + 32$	<code>Data[4][0]</code>	第 5 行
17	$X + (4 * 4 + 1) * 2$ $= X + 34$	<code>Data[4][1]</code>	
18	$X + (4 * 4 + 2) * 2$ $= X + 36$	<code>Data[4][2]</code>	
19	$X + (4 * 4 + 3) * 2$ $= X + 38$	<code>Data[4][3]</code>	

若以列为主来表示:

$\text{Data}[i][j]$ 的内存位置

$$= \text{数组第一个元素位置} + [(j * \text{每一列的元素个数}) + i] \\ * (\text{所声明数据类型所占的大小})$$

所以我们可以画出下面这个表:

	内存位置	数组名	列数
0	$X + (0 * 5 + 0) * 2$ $= X$	<code>Data[0][0]</code>	第 1 列
1	$X + (0 * 5 + 1) * 2$ $= X + 2$	<code>Data[1][0]</code>	
2	$X + (0 * 5 + 2) * 2$ $= X + 4$	<code>Data[2][0]</code>	
3	$X + (0 * 5 + 3) * 2$ $= X + 6$	<code>Data[3][0]</code>	

续表

	内存位置	数组名	行数
4	$X+(0*5+4)*2$ $=X+8$	Data[4][0]	第 2 列
5	$X+(1*5+0)*2$ $=X+10$	Data[0][1]	
6	$X+(1*5+1)*2$ $=X+12$	Data[1][1]	
⋮	⋮	⋮	⋮
15	$X+(3*5+0)*2$ $=X+30$	Data[0][3]	第 4 列
16	$X+(3*5+1)*2$ $=X+32$	Data[1][3]	
17	$X+(3*5+2)*2$ $=X+34$	Data[2][3]	
18	$X+(3*5+2)*2$ $=X+36$	Data[3][3]	
19	$X+(3*5+3)*2$ $=X+38$	Data[4][3]	

综合上列对数组的表示法的说明, 我们可整理出一个可应用的式子。

二维数组中, 我们定义出:

数组中第一个元素位置为 X , 每个元素占用的空间为 M , 每一行的元素个数为 R , 每一列的元素个数为 C

则以行为主的数组:

Data[i][j]的内存位置 = $X + [(i * R) + j] * M$

则以列为主的数组:

Data[i][j]的内存位置 = $X + [(j * C) + i] * M$

程序实例:

设计一个能将二维数组转换成以列为主的一维数组和以行为为主的一维数组的程序

默认二维数组数据为:

$$\text{Data} = \begin{bmatrix} 9 & 7 & 6 & 6 \\ 3 & 5 & 3 & 3 \\ 6 & 6 & 4 & 7 \\ 7 & 5 & 1 & 4 \\ 1 & 2 & 8 & 0 \end{bmatrix}$$

程序构思:

依题意可得, 本程序所运用的数据结构为“二维数组”, 目的为进行数组以列为主和以行为主的转换。

已知该二维数组的大小为 $5*4$ 。

以行为主的数组转换公式为:

$\text{Data}[i][j]$ 的位置 $= (i * 4) + j$

以列为主的数组转换公式为:

$\text{Data}[i][j]$ 的位置 $= (j * 5) + i$

声明一个大小为 20 的一维数组, 用来存储转换以列为主后的数据, 并声明一个大小为 20 的一维数组, 用来存储转换以行为主后的数据。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-06.c */
03  /* 程序目的: 设计一个二维矩阵转换成一维数组的程序 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  void main(void)
07  {
08      /* 默认 5*4 的矩阵数据 */
09      int Data[5][4] = { 9, 7, 6, 6,
10                        3, 5, 3, 3,
11                        6, 6, 4, 7,
12                        7, 5, 1, 4,
13                        1, 2, 8, 0 };
14      int RowData[20]; /* 存储以行为主的数组 */
15      int ColData[20]; /* 存储以列为主的数组 */
16      int i, j; /* 循环计数变量 */
17
18      /* 输出转换前二维数组的数据 */
19      printf("The Data of two dimensional array:\n");
20      for (i=0; i<5; i++)
21      {
22          for (j=0; j<4; j++)
23              printf("%4d", Data[i][j]);
24          printf("\n");
25      }
26
27      for (i=0; i<5; i++) /* 转换以行为主的一维数组 */
28          for (j=0; j<4; j++)
29              RowData[i*4+j] = Data[i][j];
30      printf("\nThe Row Major Matrix:\n");
31      for (i=0; i<20; i++) /* 输出转换以行为主后的数组数据 */
32          printf("%3d", RowData[i]);
33      printf("\n");
34
35      for (i=0; i<5; i++) /* 转换以列为主的一维数组 */
36          for (j=0; j<4; j++)
37              ColData[j*5+i] = Data[i][j];
38      printf("\nThe Column Major Matrix:\n");
39      for (i=0; i<20; i++) /* 输出转换以列为主后的数组数据 */
40          printf("%3d", ColData[i]);
41      printf("\n");
42
43  }

```

运行结果:

```

C:\DS>array-06
The Data of two dimensional array:
 9  7  6  6
 3  5  3  3

```

```

6 6 4 7
7 5 1 4
1 2 8 0
The Row Major Matrix:
9 7 6 6 3 5 3 3 6 6 4 7 7 5 1 4 1 2 8 0
The Column Major Matrix:
9 3 6 7 1 7 5 6 5 2 6 3 4 1 8 6 3 7 4 0
C:\DS>

```

练习实例:

假设有一个浮点数二维数组共有 15 列 11 行, 数据存储方式是采以行为主, 且在内存上的起始地址是 30, 试求出数组中第 3 行第 5 列的元素在内存中的地址。

练习解答:

由题意得知, 以行为主存储

$$\text{Data}[i][j] \text{ 的内存位置} = X + [(i * R) + j] * M$$

1. 浮点数二维数组 → 每个元素占用的空间为 4 个字节, $M=4$
2. 数组大小为共有 15 列 11 行 → $R=15$
3. 起始地址是 30 → $X=30$

$$\text{Data}[3][5] \text{ 的内存位置} = 30 + [(3 * 15) + 5] * 4 = 150 \quad 230$$

练习实例:

假设数组

$$\text{Demo} = \begin{bmatrix} 3 & 9 & 1 & 4 & 1 & 4 \\ 5 & 6 & 7 & 9 & 0 & 3 \\ 3 & 2 & 7 & 2 & 5 & 6 \\ 9 & 7 & 4 & 7 & 8 & 0 \end{bmatrix}$$

求出该数组以列为主和以行为主转换成一维数组后的相对位置。

练习解答:

以行为主:

一维数组中的位置	0	1	2	3	...	19	20	21	22	23
在原数组中的位置	(0,0)	(0,1)	(0,2)	(0,3)		(3, 1)	(3,2)	(3,3)	(3,4)	(3,5)
内容值	3	9	1	4	...	7	4	7	8	0

以列为主:

一维数组中的位置	0	1	2	3	...	19	20	21	22	23
在原数组中的位置	(0,0)	(1,0)	(2,0)	(3,0)		(3,4)	(0,5)	(1,5)	(2,5)	(3,5)
内容值	3	5	3	9	...	8	4	3	6	0

2.9 特殊类型的数组

本节介绍的是一些数组在数学矩阵上的特殊运用。

2.9.1 稀疏数组

首先谈到是“稀疏数组”，所谓稀疏数组就是数组中大部分的内容值都未被使用(或都为零)，在数组中仅有少部分的空间使用。因此造成内存空间的浪费，为了节省内存空间，并不影响数组中原有的内容值，我们可以采用一种压缩的方式来表示稀疏数组的内容。

有一个大小为 9×7 的数组，内容如下：

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0		0	0	0	0	0
2	0	0	0	0	0	0	0
3			0	0	0	0	0
4	0	0		0	0	0	0
5	0	0	0	0			0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0

在此数组中，共有 63 个空间，但却只使用 5 个元素，造成 58 个元素空间的浪费。以下我们就用稀疏数组重新来表示这个数组：

数组行数	数组列数	使用个数	
9	7	5	第 1 部分
1	1	3	第 2 部分
3	0	1	
3	1	4	
4	2	7	
5	5	5	
↑	↑	↑	
元素行数	元素列数	元素内容	

其中在稀疏数组中第一部分所记录的是原数组的列数和行数及元素使用的个数、第二部分所记录的是原数组中元素的位置及内容。经过压缩之后，是不是节省了不少的空间呢？原来需要声明大小为 9×7 的数组，共需 63 个存储空间，如今采用压缩后，只需要声明大小为 6×3 的数组，仅需 18 个存储空间。

程序实例:

设计一个将稀疏数组压缩的程序

默认原二维数组数据为

Data =

0	0	0	0	0	0	0
0	3	0	0	0	0	0
0	0	0	0	0	0	0
1	4	0	0	0	0	0
0	0	7	0	0	0	0
0	0	0	0	0	5	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

程序构思:

依题意可得, 我们需知道原稀疏数组的大小, 及使用的元素个数, 并声明另一个大小为 6×3 的数组来存储原数组的数据。

已知原数组的大小为 9×7 , 但不知稀疏数组中有几个元素已使用。

我们利用一个两层的循环, 遍历原数组中的每一笔数据, 当数据内容不为零时, 则记录其列数、行数及内容于新数组中。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-07.c */
03  /* 程序目的: 设计一个将稀疏数组压缩的程序 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  void main(void)
07  {
08      /* 默认 9*7 的稀疏矩阵数据 */
09      int Data[9][7] = { 0, 0, 0, 0, 0, 0, 0,
10                        0, 3, 0, 0, 0, 0, 0,
11                        0, 0, 0, 0, 0, 0, 0,
12                        1, 4, 0, 0, 0, 0, 0,
13                        0, 0, 7, 0, 0, 0, 0,
14                        0, 0, 0, 0, 0, 5, 0,
15                        0, 0, 0, 0, 0, 0, 0,
16                        0, 0, 0, 0, 0, 0, 0,
17                        0, 0, 0, 0, 0, 0, 0 };
18      int CompressData[6][3]; /* 存储压缩后数据的数组 */
19      int Index; /* 压缩数组的索引值 */
20      int i, j; /* 循环计数变量 */
21
22      Index = 0; /* 数组索引值初始化 */
23
24      printf("Two dimensional sparse array:\n");
25      for (i=0; i<9; i++) /* 输出压缩前数组数据 */
26      {
27          for (j=0; j<7; j++)
28              printf("%3d", Data[i][j]);
29          printf("\n");

```

```

30     }
31
32     for (i=0;i<9;i++) /* 进行数组数据压缩 */
33         for (j=0;j<7;j++)
34             if (Data[i][j] != 0)
35             {
36                 Index++; /* 增加数组索引值 */
37                 /* 元素行位置 */
38                 CompressData[Index][0] = i;
39                 /* 元素列位置 */
40                 CompressData[Index][1] = j;
41                 /* 元素内容值 */
42                 CompressData[Index][2] = Data[i][j];
43             }
44
45     CompressData[0][0] = 9; /* 原数组的行数 */
46     CompressData[0][1] = 7; /* 原数组的列数 */
47     CompressData[0][2] = Index; /* 使用的元素个数 */
48
49     printf("Two dimensional compress array:\n");
50     for (i=0; i<=Index;i++) /* 输出压缩后数组数据 */
51     {
52         for (j=0;j<3;j++)
53             printf("%3d",CompressData[i][j]);
54         printf("\n");
55     }
56 }

```

运行结果:

```

C:\DS>array-07
Two dimensional sparse array:
0 0 0 0 0 0 0
0 3 0 0 0 0 0
0 0 0 0 0 0 0
1 4 0 0 0 0 0
0 0 7 0 0 0 0
0 0 0 0 0 5 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0

Two dimensional compress array:
9 7 5
1 1 3
3 0 1
3 1 4
4 2 7
5 5 5

C:\DS>

```

2.9.2 上三角数组

对于一个行列个数相等的数组，我们在数学上称之为“方阵”。而所谓的“上三角数组”就是一种方阵主对角线的左下方元素全部都为0的数组。

数组 Data 的内容如下图：

	0	1	2	3	4
0	3	9	1	4	7
1	0	5	2	5	8
2	0	0	5	2	4
3	0	0	0	1	7
4	0	0	0	0	9

对于这种“上三角数组”我们发现在使用中会造成大部分存储空间的浪费。所以我们可以思考如何以一种有利且节省空间的方式来存储这类数组。还记得我们曾在上一节提到数组的表示法，并说明二维数组转换成一维数组的方式，基于此种构想，我们便可以将“上三角数组”转换成一维数组，以节省空间。

以行为主：

如果将一个大小为 $n \times n$ 的上三角数组转换成以行为主的一维数组，且不存储内容为 0 的元素，我们可以得到：

	0	1	2	...	j	...	n
0							
1	0						
2	0	0					
⋮	⋮	⋮					
i	0	0		...	Data[i][j]		
⋮	⋮	⋮	⋮				
n	0	0	0	0	0	0	

$\text{Data}[i][j]$ 的位置 = 花纹部分 + 黑色部分 = $[(n+1) + (n-i+1)] * i / 2 + (j-i)$

一维数组中的位置	0	1	2	3	4	5	6	...	10	11	12	13	14
在原数组中的位置	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(1,1)	(1,2)	...	(2,3)	(2,4)	(3,3)	(3,4)	(4,4)
内容值	3	9	1	4	7	5	2	...	2	4	1	7	9
行数	第 1 行				第 2 行				第 3 行		第 4 行		第 5 行

程序实例：

设计一个将上三角数组转换成以行为主一维数组的程序

默认原上三角数组数据为

$$\text{Upper} = \begin{bmatrix} 3 & 9 & 1 & 4 & 7 \\ 0 & 5 & 2 & 5 & 8 \\ 0 & 0 & 5 & 2 & 4 \\ 0 & 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 0 & 9 \end{bmatrix}$$

程序构思:

依题意可知, 上三角数组大小为 5×5 , 我们需要声明一个大小为 15 的数组来存储原数组的数据, $5 \times (1+5)/2$ 得 15。

我们利用一个两层的循环遍历原数组中的每一笔数据; 当元素的行数小于列数时, 则将该元素存储于一维数组的 $[(n+1)+(n-i+1)] \times i/2 + (j-i)$ 位置。

已知数组大小为 5, 故 $n=5$ 代入得转换公式: $(11-i) \times i/2 + (j-i)$ 。

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: array-08.c */
03  /* 程序目的: 将上三角数组转换成以行为主的一维数组 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  void main(void)
07  {
08      /* 默认 5*5 的上三角矩阵数据 */
09      int Upper[5][5] = { 3, 9, 1, 4, 7,
10                          0, 5, 2, 5, 8,
11                          0, 0, 5, 2, 4,
12                          0, 0, 0, 1, 7,
13                          0, 0, 0, 0, 9 };
14      int RowMajor[15]; /* 存储转换后数据的数组 */
15      int Index; /* 数组的索引值 */
16      int i, j; /* 循环计数变量 */
17
18      printf("Two dimensional upper triangular array:\n");
19
20      for (i=0; i<5; i++) /* 输出上三角数组数据 */
21      {
22          for (j=0; j<5; j++)
23              printf("%3d", Upper[i][j]);
24          printf("\n");
25      }
26
27      for (i=0; i<5; i++) /* 进行数组数据转换 */
28      {
29          for (j=0; j<5; j++)
30              if (i <= j)
31              {
32                  Index = (11-i)*i/2 + (j-i);
33                  RowMajor[Index] = Upper[i][j];
34              }
35          printf("\n");
36
37          printf("Row Major one dimensional array:\n");
38          for (i=0; i<15; i++) /* 输出转换后数组数据 */
39              printf("%3d", RowMajor[i]);
40          printf("\n");
41      }
42  }
```

运行结果:

```
C:\DS>array-08
Two dimensional upper triangular array:
 3  9  1  4  7
```

```

0 5 2 5 8
0 0 5 2 4
0 0 0 1 7
0 0 0 0 9
Row Major one dimensional array:
3 9 1 4 7 5 2 5 8 5 2 4 1 7 9

C:\DS>

```

以列为主:

如果将一个大小为 $n \times n$ 的上三角数组转换成以列为主的一维数组, 且不存储内容为 0 的元素, 我们可以得到:

	0	1	2	...	j	...	n
0							
1	0						
2	0	0					
⋮	⋮	⋮					
i	0	0				...	
⋮	⋮	⋮					
N	0	0	0	0	0	0	

$\text{Data}[i][j]$ 的位置 = 花纹部分 + 黑色部分 = $j * (j + 1) / 2 + i$

一维数组中的位置	0	1	2	3	4	5	6	...	10	11	12	13	14
在原数组中的位置	(0,0)	(0,1)	(1,1)	(0,2)	(1,2)	(2,2)	(0,3)	...	(0,4)	(1,4)	(2,4)	(3,4)	(4,4)
内容值	3	9	5	1	2	5	4		7	8	4	7	9
列数	第 1 列	第 2 列		第 3 列			第 4 列			第 5 列			

程序实例:

设计一个将上三角数组转换成以列为主一维数组的程序

默认原上三角数组数据为

$$\text{Upper} = \begin{bmatrix} 3 & 9 & 1 & 4 & 7 \\ 0 & 5 & 2 & 5 & 8 \\ 0 & 0 & 5 & 2 & 4 \\ 0 & 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 0 & 9 \end{bmatrix}$$

程序构思:

依题意可知, 上三角数组大小为 5×5 , 我们需要声明一个大小为 15 的数组来存储原数组的数据, $5 * (1 + 5) / 2$ 得 15。

我们利用一个两层的循环遍历原数组中的每一笔数据, 当元素的行数小于列数时, 则将该元素存储于一维数组的 $j * (j + 1) / 2 + i$ 位置。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-09.c */
03  /* 程序目的: 将上三角数组转换成以列为主的一维数组 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  void main(void)
07  {
08      /* 默认 5*5 的上三角矩阵数据 */
09      int Upper[5][5] = { 3, 9, 1, 4, 7,
10      0, 5, 2, 5, 8,
11      0, 0, 5, 2, 4,
12      0, 0, 0, 1, 7,
13      0, 0, 0, 0, 9 };
14      int ColMajor[15]; /* 存储转换后数据的数组 */
15      int Index; /* 数组的索引值 */
16      int i, j; /* 循环计数变量 */
17
18      printf("Two dimensional upper triangular array:\n");
19
20      for (i=0; i<5; i++) /* 输出上三角数组数据 */
21      {
22          for (j=0; j<5; j++)
23              printf("%3d", Upper[i][j]);
24          printf("\n");
25      }
26
27      for (i=0; i<5; i++) /* 进行数组数据转换 */
28      {
29          for (j=0; j<5; j++)
30              if (i <= j)
31              {
32                  Index = j*(j+1)/2 + i;
33                  ColMajor[Index] = Upper[i][j];
34              }
35          printf("\n");
36      }
37      printf("Column Major one dimensional array:\n");
38      for (i=0; i<15; i++) /* 输出转换后数组数据 */
39          printf("%3d", ColMajor[i]);
40      printf("\n");
41  }

```

运行结果:

```

C:\DS>array-09
Two dimensional upper triangular array:
  3  9  1  4  7
  0  5  2  5  8
  0  0  5  2  4
  0  0  0  1  7
  0  0  0  0  9

Column Major one dimensional array:
  3  9  5  1  2  5  4  5  2  1  7  8  4  7  9

C:\DS>

```

2.9.3 下三角数组

介绍了上三角数组后，接下来我们再介绍下三角数组。所谓的“下三角数组”就是一种方阵主对角线的右下方元素全部都为 0 的数组。

数组 Data 的内容如下图：

	0	1	2	3	4
0	3	0	0	0	0
1	7	5	0	0	0
2	6	4	5	0	0
3	8	3	2	1	0
4	9	1	6	4	9

以行为主：

如果将一个大小为 $n \times n$ 的下三角数组转换成以行为主的一维数组，且不存储内容为 0 的元素，我们可以得到：

	0	1	2	...	j	...	n
0		0	0				0
1			0				0
2							0
⋮					⋮		
i				...	Data[i][j]		0
⋮							
N							

Data[i][j]的位置= 花纹部分 + 黑色部分= $i*(i+1)/2 + j$

一维数组中的位置	0	1	2	3	4	5	6	...	10	11	12	13	14
在原数组中的位置	(0,0)	(1,0)	(1,1)	(2,0)	(2,1)	(3,0)	(3,1)		(4,0)	(4,1)	(4,2)	(4,3)	(4,4)
内容值	3	7	5	6	4	5	8		9	1	6	4	9
行数	第 1 行	第 2 行	第 3 行	第 4 行	第 5 行				第 10 行	第 11 行	第 12 行	第 13 行	第 14 行

程序实例：

设计一个将下三角数组转换成以行为主一维数组的程序
默认原下三角数组数据为

Lower =
$$\begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 7 & 5 & 0 & 0 & 0 \\ 6 & 4 & 5 & 0 & 0 \\ 8 & 3 & 2 & 1 & 0 \\ 9 & 1 & 6 & 4 & 9 \end{bmatrix}$$

程序构思:

依题意得知, 下三角数组大小为 5×5 , 我们需要声明一个大小为 15 的数组来存储原数组的数据, $5 \times (1+5)/2 = 15$ 。

我们利用一个两层的循环遍历原数组中的每一笔数据, 当元素的行数大于列数时, 则将该元素存储于一维数组的 $i \times (i+1)/2 + j$ 位置。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-10.c */
03  /* 程序目的: 将下三角数组转换成以行为主的一维数组 */
04  /* Written By Kuo-Yu Huang.. (WANT Studio.) */
05  /* ===== */
06  void main(void)
07  {
08      /* 默认 5*5 的下三角矩阵数据 */
09      int Lower[5][5] = { 3, 0, 0, 0, 0,
10      7, 5, 0, 0, 0,
11      6, 4, 5, 0, 0,
12      8, 3, 2, 1, 0,
13      9, 1, 6, 4, 9, };
14      int RowMajor[15]; /* 存储转换后数据的数组 */
15      int Index; /* 数组的索引值 */
16      int i, j; /* 循环计数变量 */
17
18      printf("Two dimensional Lower triangular array:\n");
19
20      for (i=0; i<5; i++) /* 输出下三角数组数据 */
21      {
22          for (j=0; j<5; j++)
23              printf("%3d", Lower[i][j]);
24          printf("\n");
25      }
26
27      for (i=0; i<5; i++) /* 进行数组数据转换 */
28      {
29          for (j=0; j<5; j++)
30              if (i >= j)
31              {
32                  Index = i*(i+1)/2 + j;
33                  RowMajor[Index] = Lower[i][j];
34              }
35          printf("\n");
36      }
37      printf("Row Major one dimensional array:\n");
38      for (i=0; i<15; i++) /* 输出转换后数组数据 */
39          printf("%3d", RowMajor[i]);
40      printf("\n");
41  }

```

运行结果:

```

C:\DS>array-10
Two dimensional Lower triangular array:
 3 0 0 0 0
 7 5 0 0 0
 6 4 5 0 0
 8 3 2 1 0
 9 1 6 4 9

```

```

Row Major one dimensional array:
3 7 5 6 4 5 8 3 2 1 9 1 6 4 9
C:\DS>

```

以列为主:

如果将一个大小为 $n \times n$ 的下三角数组转换成以列为主的一维数组, 且不存储内容为 0 的元素, 我们可以得到:

	0	1	2	...	j	...	n
0		0	0				0
1			0				0
2							0
⋮							
i					Data[i][j]		0
⋮							
n							

$\text{Data}[i][j]$ 的位置 = 花纹部分 + 黑色部分 = $[(n+1)+(n-j+1)] * j / 2 + (i-j)$

一维数组 中的位置	0	1	2	3	4	5	6	...	10	11	12	13	14
在原数组 中的位置	(0,0)	10	20	30	40	11	21	...	32	42	33	43	44
内容值	3	7	6	8	9	5	4	...	2	6	1	4	9
列数	第 1 列					第 2 列			第 3 列		第 4 列		第 5 列

程序实例:

设计一个将下三角数组转换成以行为主一维数组的程序
默认原下三角数组数据为

$$\text{Lower} = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 7 & 5 & 0 & 0 & 0 \\ 6 & 4 & 5 & 0 & 0 \\ 8 & 3 & 2 & 1 & 0 \\ 9 & 1 & 6 & 4 & 9 \end{bmatrix}$$

程序构思:

我们利用一个两层的循环遍历原数组中的每一笔数据, 当元素的行数大于列数时, 则将该元素存储于一维数组的 $[(n+1)+(n-j+1)] * j / 2 + (i-j)$ 位置。已知数组大小为 5, 故 $n=5$ 代入得转换公式: $(11-j) * j / 2 + (i-j)$ 。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: array-11.c */

```

```

03  /* 程序目的: 将下三角数组转换成以列为主的一维数组          */
04  /* Written By Kuo-Yu Huang. (WANT Studio.)                  */
05  /* =====                                                  */
06  void main(void)
07  {
08      /* 默认 5*5 的下三角矩阵数据*/
09      int    Lower[5][5] = { 3, 0, 0, 0, 0,
10                          7, 5, 0, 0, 0,
11                          6, 4, 5, 0, 0,
12                          8, 3, 2, 1, 0,
13                          9, 1, 6, 4, 9,};
14      int    ColMajor[15];      /* 存储转换后数据的数组 */
15      int    Index;            /* 数组的索引值 */
16      int    i,j;              /* 循环计数变量 */
17
18      printf("Two dimensional Lower triangular array:\n");
19
20      for (I=0;i<5;i++)        /* 输出下三角数组数据 */
21      {
22          for (j=0;j<5;j++)
23              printf("%3d",Lower[i][j]);
24          printf("\n");
25      }
26
27      for (I=0;i<5;i++)        /* 进行数组数据转换 */
28      for (j=0;j<5;j++)
29          if (i >= j)
30          {
31              Index = (11.j)*j/2 + (i-j);
32              ColMajor[Index] = Lower[i][j];
33          }
34      printf("\n");
35
36      printf("Column Major one dimensional array:\n");
37      for (I=0;i<15;i++)        /* 输出转换后数组数据 */
38          printf("%3d",ColMajor[I]);
39      printf("\n");
40  }

```

运行结果:

```

C:\DS>array-11
Two dimensional Lower triangular array:
 3 0 0 0 0
 7 5 0 0 0
 6 4 5 0 0
 8 3 2 1 0
 9 1 6 4 9
Column Major one dimensional array:
 3 7 6 8 9 5 4 3 1 5 2 6 1 4 9
C:\DS>

```

【习题】

一、复习:

1. 数组所使用的是一种静态的内存空间配置?
2. 是否可以将多个不同数据类型的数据存储于同一个数组中?

3. 如果声明一个大小为 20 的整数数组, 是否一定要在数组中 20 个元素全存放数据?
4. 数组的表示法, 有以行为主和以列为主两种。
5. 若使用以行为主的数组表示法比以列为主的数组表示法, 节省空间。
6. 在 C 语言中声明一个数组为: `int Data[20]`
则其可使用的空间为 `Data[1]~Data[20]`。
7. 下列叙述哪一个为非?
 - (a) 每一个数组, 都有一个索引值和一个内容值。
 - (b) 索引值是用来方便存取数据。
 - (c) 内容值正是该索引值被存储数据的位置。
 - (d) C 语言中的数组索引值从 0 开始。
8. 下列哪一个为声明一个大小为 30 的字符数组的方法?
 - (a) `char Data[29];`
 - (b) `char Data[30];`
 - (c) `char Data[31];`
 - (d) 以上皆非。
9. 若有一个整数数组 X 是采用以行为主的表示法, 已知其 `X[3,5]` 的地址为 1000, `X[5,7]` 的地址为 1200, 则下列语句哪一个不正确?
 - (a) 行的个数为 49。
 - (b) `X[8,10]` 的地址为 1600。
 - (c) `X[3,8]` 的地址为 1016。
 - (d) 整数数组每一个元素占 2 个 bytes。

二、应用:

1. 试说明何谓“静态内存配置”? 其优、缺点为何?
2. 若声明一个浮点数数组如下:

```
float Average[30];
```

假设该数组的内存起始位置为 200, 试求 `Average[15]` 和 `Average[27]` 的内存地址?

3. 试利用 C 语言写出在数组中插入元素和删除元素的子程序。
4. 试说明何谓以列为主和以行为主。
5. 试利用以行为主的方法, 来说明数组

```

9   7   5   0   1
3   5   4   6   8
1   8   2   5   4
3   0   7   1   8
9   5   1   8   6

```

在内存中的存储方式。

6. 若有一个二维数组 Data, 排列方式为 `Data[3][5]` 的内存地址为 3000, `Data[4][6]` 的内存地址为 3600, 试求 `Data[5][7]` 的内存地址。
7. 假设有一个浮点数二维数组共有 7 行 9 列, 数据存储方式是采以行为主, 且在内存上的起始地址是 300, 试求出数组中第 6 行第 5 列的元素在内存中的地址。
8. 试求出稀疏数组

```

00   1   0   8   0
00   0   0   0   0

```


04	0	3	0	0
60	0	0	0	0
00	0	0	0	0
05	0	0	0	0
00	0	0	0	0

压缩后的数组内容。

9. 若有一个大小为 5×5 的上三角数组，试求出数组(3,4)以列为主和以行为主转换后的一维数组索引值。

链 表

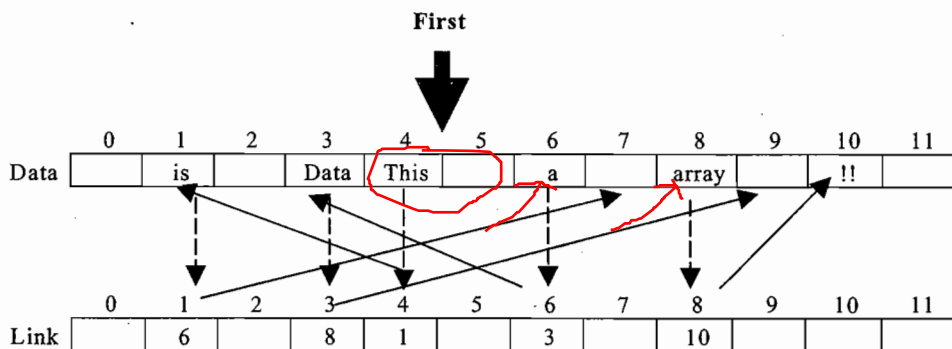
第 3 章



- ◆ 何谓链表
- ◆ 单链表的建立
- ◆ 单链表的基本处理

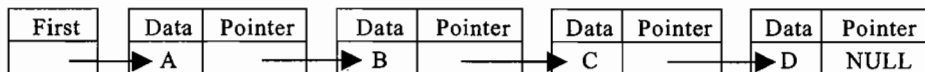
3.1 何谓链表

何谓链表？链表(Linked List)是一种有序的列表(Ordered List)，链表的内容通常是存储于内存中分散的位置上。链表串连的方式有两种：一种是利用数组结构串连的有序列表。如利用两个数组，一个存放数据，另一个存放链接的关系，假设是从数组中的 Data[4]元素开始，则数据的链接关系如下图：



不过这种列表最大的缺点是在放插入或删除元素时，常常需要大量的搬动其它元素，而且数组的大小是固定的，缺乏使用弹性。

另一种是以动态内存配置的链表，通常我们所指的“链表”如果没有特别说明，就是指以动态内存配置的链表，本章所谈的，也是针对这种链表做说明。一个动态内存配置的链表，是由许许多多的节点(Node)所链接而成的，每一个节点，包含了数据部分和指向链表中下一个节点的指针(Pointer)。以动态内存配置的链表，在插入或删除元素时，只需要将指针改变指向即可。链表的结构如下图所示。



上图中的指针(→)就是链接(Link)。而最后一个的指针是 NULL，表示这是列表的结尾。

3.2 单链表的建立

3.2.1 单链表内节点的配置

因为链表是一种动态配置的内存空间，也就是说在程序运行的过程中才向系统配置所需的内存空间。上一节我们提到一个动态内存配置的链表，是由许许多多的节点所链接而成的，每一个节点，包含了数据部分和指向链表中下一个节点的指针。所以在使用链表前，我们必须先定义出节点的数据结构。在 C 语言中，节点结构的声明格式如下：

```
struct    结构名称
{
    数据类型    数据变量;
    struct      List    *Next;
};
```

```
typedef struct 结构名称 Node;
typedef Node *Link;
```

如果我们以后在程序中要使用此结构只需要声明为:

```
Link 变量名称
```

例如, 我们现在有一个学生数据, 包括了学生学号、学生姓名、学生电话, 我们使用链表来定义此结构:

```
struct List
{
    int    Number;
    char   Name[10];
    char   Telephone[12];
    struct List *Next;
};
typedef struct List Node;
typedef Node *Link;
```

在程序中, 若我们声明为:

```
Link Student; //Student 变量为之前所声明的数据结构
```

当然声明出以上的数据结构, 程序并不会马上去配置我们所需要的内存空间, 而是需要利用 `malloc` 函数向系统要求配置内存空间, 对于一个结构如下:

```
struct List
{
    int    Data;
    struct List *Next;
};
typedef struct List Node;
typedef Node *Link;
```

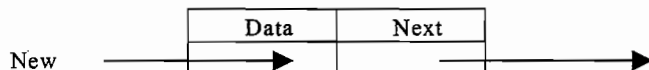
```
Link New;
```

则 `malloc` 函数使用的方法为:

```
New = (Link) malloc(sizeof( Node));
```

当内存配置成功时, `New` 所返回的将是一个指针, 当内存配置不成功时, `New` 所返回的则是 `NULL` 指针。(因为 `malloc` 函数是在 `stdlib` 内, 所以在使用 `malloc` 时, 必须在程序开头: `#include <stdlib.h>`)

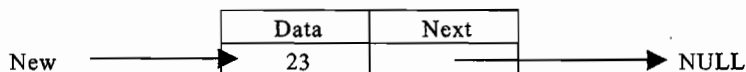
如果以上程序内存配置成功的话, 其结构如下:



如果我们要将这个节点的数据设为 23, 将 `Next` 设为 `NULL`, 我们可以用:

```
New->Data = 23;
New->Next = NULL;
```

则此时数据的结构如下:



3.2.2 单链表内节点的释放

利用 malloc 函数向系统要求配置内存空间后, 若没有将这种内存空间释还给系统, 这个内存空间将会持续的占住内存, 当使用很多的动态内存配置之后, 可能发生内存不足的问题, 所以当这个内存空间不再使用的时候, 我们最好能将内存空间给释放。在 C 语言中, 释放内存空间的是 free 函数, 其声明格式如下:

```
free(变量名称);
```

例如, 有一个动态内存的配置如下:

```
struct List
{
    int Data;
    struct List *Next;
};
typedef struct List Node;
typedef Node *Link;
void main()
{
    Link New;
    New = (Link) malloc(sizeof( Node));
    : //略
    : //略
}
```

当我们需要释放内存空间时, 我们只需要声明:

```
free(New);
```

程序实例:

设计一个节点的配置与释放的程序。

程序构思:

节点的结构:

```
struct List
{
    int Data;
    struct List *Next;
};
typedef struct List Node;
```

```
typedef Node *Link;
```

节点的配置:

```
New = (Link) malloc(sizeof( Node));
```

节点的释放:

```
free(New);
```

程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: list.c */
03 /* 程序目的: 设计一个节点的配置与释放的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define Max 10
08 struct List /* 节点结构声明 */
```

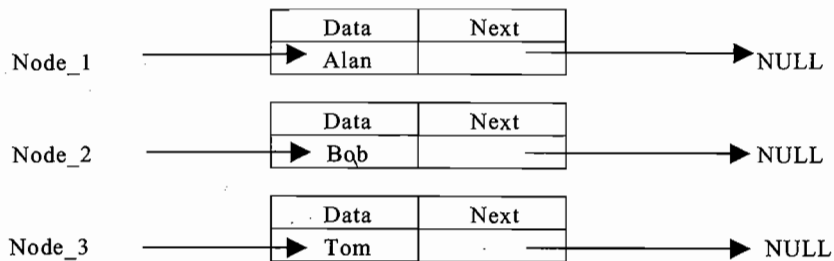
```
09 {
10     int    Number;
11     char   Name[Max];
12     struct List *Next;
13 };
14 typedef struct List Node;
15 typedef Node *Link;
16
17 void main ()
18 {
19     Link   New;           /* 节点声明 */
20     int    DataNum;       /* 数据编号 */
21     char   DataName[Max]; /* 数据名称 */
22     int    i;
23
24     New = (Link) malloc(sizeof(Node)); /* 内存配置 */
25
26     if ( New == NULL )
27         printf("Memory allocate Failure!!\n"); /* 内存配置失败 */
28
29     else
30     {
31         printf("Please input the data number : ");
32         scanf("%d",&DataNum);
33         printf("Please input the data name : ");
34         scanf("%s",DataName);
35
36         New->Number = DataNum;
37         for ( i=0;i<=Max;i++ )
38         {
39             New->Name[i] = DataName[i];
40         }
41         New->Next = NULL;
42
43         printf("##Input Data##\n");
44         printf("Data Number : %d\n",New->Number);
45         printf("Data Name : %s\n",New->Name);
46     }
47     free(New); /* 内存释放 */
48 }
```

运行结果:

```
C:\DS>list
Please input the data number : 256
Please input the data name : Computer
##Input Data##
Data Number : 256
Data Name : Computer
C:\DS>
```

3.2.3 单链表的建立与释放

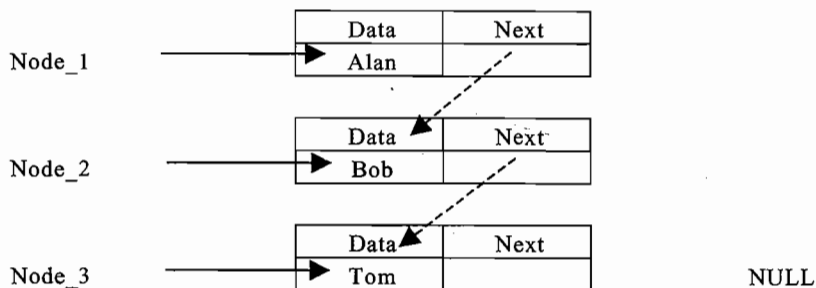
单链表的建立是将每一个节点单向地串连起来。例如：现在有 3 个节点如下：



我们现在想要把 3 个节点依次给串连起来, 则运作过程如下:

```

Node_1->Next = Node_2
Node_2->Next = Node_3
  
```



单链表则需要一个一个的将节点释放, 直到下一个节点指针指向 NULL 为止。如上图, 链表释放时, 先将 Pointer 指针指向第一个节点, 将 Pointer(即第一个节点)释放后, 再将 Pointer 指针指向 Pointer 指针的指针(即下一节点), 重复执行上一个步骤直到 Pointer 指针指向 NULL 为止。

程序实例:

设计一个将输入的数据建立成链表、输出链表数据, 并在程序结束后释放。

程序构思:

链表的建立:

先声明一个首节点 Head, 并将 Head->Next 设为 NULL。

每输一笔数据就声明一个新节点 New, 把 New->Next 设为 NULL, 并且链接到之前列表的尾端。

链表数据的输出:

先将 Pointer 节点的指针指向第一个节点, 将 Pointer 节点(即第一个节点)的数据输出。

然后再将 Pointer 节点的指针指向 Pointer 指针的指针(即下一节点), 将 Pointer 节点(即第一个节点)的数据输出。重复执行此步骤直到 Pointer 指针指向 NULL 为止。

链接的释放:

先将 Pointer 节点的指针指向第一个节点, 然后再将首节点设为首节点的指针(即下一节点), 将 Pointer 节点(即第一个节点)释放。重复执行此步骤直到首节点的指针指向 NULL 为止。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: create.c */
03  /* 程序目的: 设计一个将输入的数据建立成链表的程序 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
  
```

```
06  #include <stdlib.h>
07  #define Max 10
08  struct    List          /* 节点结构声明 */
09  {
10      int    Number;
11      char   Name[Max];
12      struct List *Next;
13  };
14  typedef    struct    List Node;
15  typedef    Node *Link;
16  /* -----*/
17  /* 释放链表 */
18  /* -----*/
19  void Free_List(Link Head)
20  {
21      Link  Pointer;      /* 节点声明 */
22
23      while (    Head != NULL ) /* 当节点为 NULL 结束循环 */
24      {
25          Pointer = Head;
26          Head = Head->Next; /* 往下一个节点 */
27          free(Pointer);
28      }
29  }
30
31  /* -----*/
32  /* 输出链表数据 */
33  /* -----*/
34  void Print_List(Link Head)
35  {
36      Link  Pointer;      /* 节点声明 */
37      Pointer = Head;      /* Pointer 指针设为首节点 */
38      while (    Pointer != NULL ) /* 当节点为 NULL 结束循环 */
39      {
40          printf("##Input Data##\n");
41          printf("Data Number : %d\n",Pointer->Number);
42          printf("Data Name : %s\n",Pointer->Name);
43          Pointer = Pointer->Next; /* 往下一个节点 */
44      }
45  }
46
47  /* -----*/
48  /* 建立链表 */
49  /* -----*/
50  Link Create_List(Link Head)
51  {
52      int    DataNum;      /* 数据编号 */
53      char   DataName[Max]; /* 数据名称 */
54      Link   New;          /* 节点声明 */
55      Link   Pointer;      /* 节点声明 */
56      int    i;
57
58      Head = (Link) malloc(sizeof(Node)); /* 内存配置 */
59
60      if ( Head == NULL )
61          printf("Memory allocate Failure!!\n"); /* 内存配置失败 */
62      else
63      {
64          DataNum = 1;      /* 初始数据编号 */
65          printf("Please input the data name : ");
66          scanf("%s",DataName);
```



```

67
68     HeadD->Number = DataNum;           /* 定义首节点数据
69     编号 */
70
71     for ( i=0;i<=Max;i++ )
72         HeadD->Name[i] = DataName[i];
73
74     HeadD->Next = NULL;
75
76     Pointer = Head;                     /* Pointer 指针设
77     为首节点 */
78
79     while (1)
80     {
81         DataNum++;                      /* 数据编号递增 */
82         New = (Link) malloc(sizeof(Node)); /* 内存配置 */
83         printf("Please input the data name : ");
84         scanf("%s",DataName);
85         if ( DataName[0] == '0' ) /* 输入 0 则结束输入 */
86
87             break;
88         New->Number = DataNum;
89         for ( i=0;i<=Max;i++ )
90         {
91             New->Name[i] = DataName[i];
92         }
93         New->Next = NULL;
94
95         Pointer->Next = New;             /* 将新节点串连在原
96         列表末端 */
97
98         Pointer = New;                  /* 列表尾端节点为新
99         节点 */
100     }
101 }
102 return Head;
103 }
104
105 /* ----- */
106 /* 主程序 */
107 /* ----- */
108 void main ()
109 {
110     Link  Head;                        /* 节点声明 */
111
112     Head = Create_List(Head);          /* 调用建立链表 */
113
114     If ( Head != NULL )
115     {
116         Print_List(Head);              /* 调用输出链表数据 */
117         Free_List(Head);               /* 调用释放链表 */
118     }
119 }

```

运行结果:

```

C:\DS>create
Please input the data name : Bob
Please input the data name : Alan
Please input the data name : Tom

```

```

Please input the data name : Judy
Please input the data name : 0
##Input Data##
Data Number : 1
Data Name : Bob
##Input Data##
Data Number : 2
Data Name : Alan
##Input Data##
Data Number : 3
Data Name : Tom
##Input Data##
Data Number : 4
Data Name : Judy
C:\DS>

```

3.2.4 单链表的查找

在单链表中的数据查找，只能采用线性查找法往下一个节点查找。

程序实例：

设计一个查找链表中数据的程序。

程序构思：

采用线性查找法查找链表中的数据。和数组不同的是，原来数组是用递增数组索引来查找数据，在链表中是往下一个节点查找。

程序源代码：

```

01  /* ===== Program Description ===== */
02  /* 程序名称:  search.c                                */
03  /* 程序目的:  设计一个查找链表中数据的程序。        */
04  /* Written By Kuo-Yu Huang. (WANT Studio.)          */
05  /* ===== */
06  #include <stdlib.h>
07  #define Max 10
08
09  struct      List                /* 节点结构声明 */
10  {
11      int      Number;
12      int      Total;
13      struct List *Next;
14  };
15  typedef      struct      List Node;
16  typedef      Node *Link;
17
18  int Data[2][Max] = /* 输入数据 */
19  {
20      { 3, 9, 25, 5, 7, 26, 65, 80, 2,
21        6, 1050, 3850, 1000, 5670, 2250, 9650, 2380, 1700, 3000, 2000 };
22
23      int SearchTime = 0; /* 查找次数 */
24      /* ----- */
25      /* 链表查找 */
26      /* ----- */
27      int List_Search(int Key, Link Head)
28      {
29          Link  Pointer;

```

```

29
30     Pointer = Head;          /* Pointer 指针设为首节点 */
31     While (    Pointer != NULL )    /* 当节点为 NULL 结束循环 */
32     {
33         SearchTime ++;
34         if ( Pointer->Number == Key )
35         {
36             printf("Data Number : %d\n",Pointer->Number);
37             printf("Data Total : %d\n",Pointer->Total);
38             return 1;
39         }
40         Pointer = Pointer->Next;    /* 往下一个节点 */
41     }
42     return 0;
43 }
44 /* ----- */
45 /* 释放链表 */
46 /* ----- */
47 void Free_List(Link Head)
48 {
49     Link    Pointer;          /* 节点声明 */
50
51     While (    Head != NULL )    /* 当节点为 NULL 结束循环 */
52     {
53         Pointer = Head;
54         Head = Head->Next;    /* 往下一个节点 */
55         free(Pointer);
56     }
57 }
58
59 /* ----- */
60 /* 建立链表 */
61 /* ----- */
62 Link Create_List(Link Head)
63 {
64     Link    New;              /* 节点声明 */
65     Link    Pointer;          /* 节点声明 */
66     int     i;
67
68     Head = (Link) malloc(sizeof(Node));    /* 内存配置 */
69
70     if ( Head == NULL )
71         printf("Memory allocate Failure!!\n"); /* 内存配置失败 */ else
72     {
73         Head->Number = Data[0][0]; /* 定义首节点数据编号 */
74         Head->Total = Data[1][0];
75         Head->Next = NULL;
76
77         Pointer = Head;          /* Pointer 指针设为首节点 */
78
79         for ( i=1;i<Max;i++)
80         {
81             New = (Link) malloc(sizeof(Node)); /* 内存配置 */
82
83             New->Number = Data[0][i];
84             New->Total = Data[1][i];
85             New->Next = NULL;
86
87             Pointer->Next = New; /* 将新节点串连在原列表尾端 */
88             Pointer = New;      /* 列表尾端节点为新节点 */
89         }

```

```
90     }
91     return Head;
92 }
93 /* ----- */
94 /* 主程序 */
95 /* ----- */
96 void main ()
97 {
98     Link    Head;          /* 节点声明 */
99     int     Num;           /* 欲查找数据编号 */
100
101     Head = Create_List(Head); /* 调用建立链表 */
102
103     if ( Head != NULL )
104     {
105
106         printf("Please input the data number : ");
107         scanf("%d",&Num);
108
109         if ( List_Search(Num,Head) )
110             printf("Search Time = %d\n",SearchTime);
111         else
112             printf("Not Found !!\n");
113         Free_List(Head);    /* 调用释放链表 */
114     }
115 }
```

运行结果:

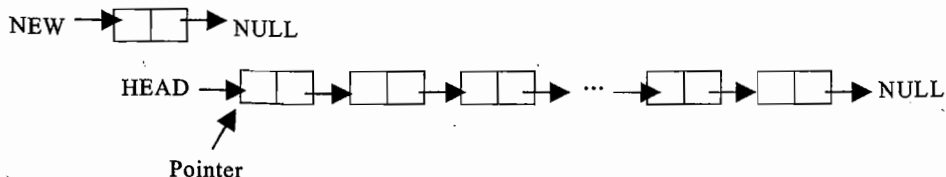
```
C:\DS>search
Please input the data number : 80
Data Number : 80
Data Total : 1700
Search Time = 8
C:\tc>search
Please input the data number : 50
Not Found !!
C:\DS>
```

3.3 单链表的基本处理

3.3.1 单链表内节点的插入

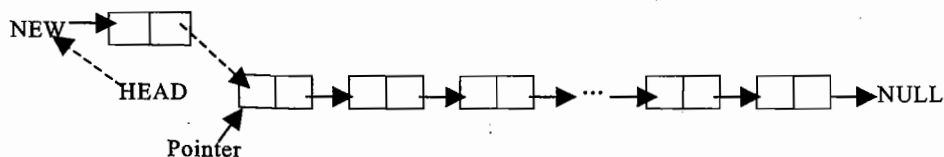
经过前两个小节的介绍之后，相信读者对链表已有基本的认识了。接下来，我们就开始介绍链表中的基本处理。首先我们要谈的是链表内节点的插入，一般来说插入节点可分为下列几类：

1. 插入在链表开头

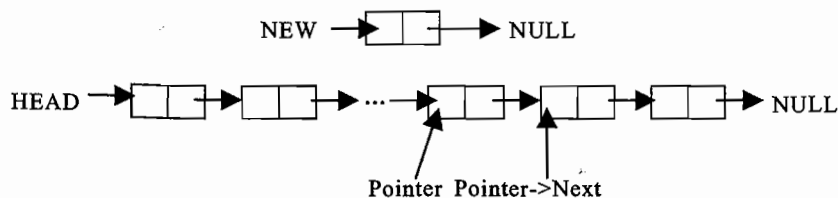


新的节点插入到链表的开头, 需要将新节点的指针指向链表的首节点, 并将链表的首节点设为新节点。

```
NEW->Next = Pointer
HEAD = New
```

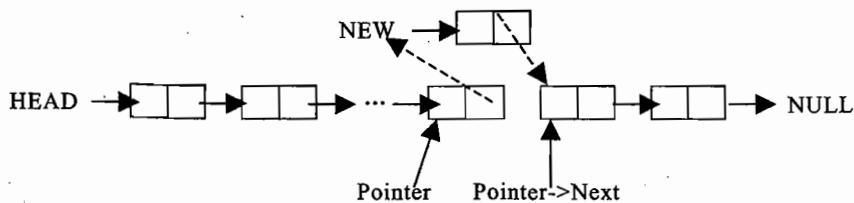


2. 插入在链表中间

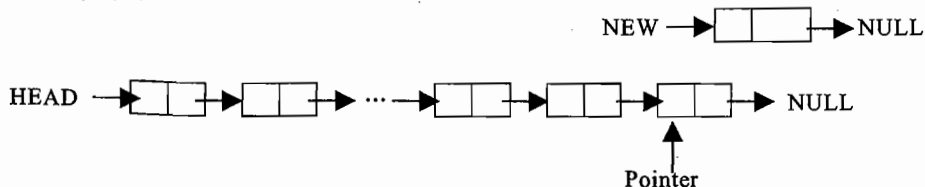


新的节点插入到链表的中间, 如果我们找到 `Pointer` 节点, 则需要将新节点的指针指向 `Pointer` 节点的指针(即下一个节点), 但不能让链表断裂。所以第 1 步必须将新节点的指针指向 `Pointer` 节点的指针, 第 2 步再将 `Pointer` 节点的指针指向新节点。

```
NEW->Next = Pointer->Next
Pointer->Next = NEW
```

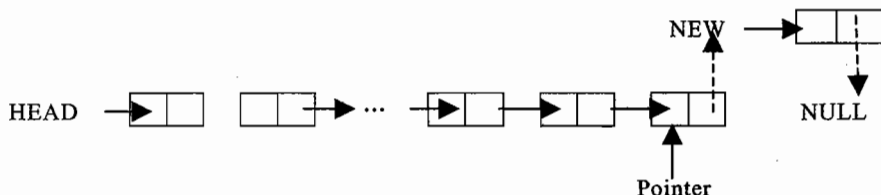


3. 插入在链表尾端



新的节点插入到链表的尾端(`Pointer` 节点), 所以第 1 步必须将新节点的指针指向 `Pointer` 节点的指针 (`NULL`), 第 2 步再将 `Pointer` 节点的指针指向新节点。

```
NEW->Next = Pointer->Next
Pointer->Next = NEW
```



程序实例:

设计一个链表内节点插入的程序。

程序构思:

声明一个新节点供用户输入欲插入节点的内容。

用户输入一个节点内容(Key), 表示欲插入在那一个节点之后。

持续往下一个节点, 直到节点内容等于 Key 或节点指针为 NULL(即找不到该节点)。

如果该节点不存在, 则插入在首节点前:

```
New->Next = Head;
Head = New;
```

如果找到该节点, 则:

```
NEW->Next = Pointer->Next
Pointer->Next = NEW
```

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: insert.c */
03  /* 程序目的: 设计一个链表内节点插入的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  #define Max 10
08
09  struct List /* 节点结构声明 */
10  {
11      int Number;
12      int Total;
13      struct List *Next;
14  };
15  typedef struct List Node;
16  typedef Node *Link;
17
18  int Data[2][Max] = /* 输入数据 */
19      { 1, 3, 5, 7, 2, 4, 6, 8, 9, 0,
20        15, 35, 10, 67, 25, 65, 38, 70, 30, 20 };
21
22  /* ----- */
23  /* 插入节点至链表内 */
24  /* ----- */
25  Link Insert_List(Link Head, Link New, int Key)
26  {
27      Link Pointer; /* 节点声明 */
28
```

```

29     Pointer = Head;          /* Pointer 指针设为首节点 */
30
31     While ( 1 )
32     {
33         if ( Pointer == NULL )    /* 插入在首节点前 */
34         {
35             New->Next = Head;
36             Head = New;
37             break;
38         }
39         if ( Pointer->Number == Key ) /* 插入在链表中或尾端 */
40         {
41             New->Next = Pointer->Next;
42             Pointer->Next = New;
43             break;
44         }
45         Pointer = Pointer->Next;    /* 往下一个节点 */
46     }
47     return Head;
48 }
49
50 /* ----- */
51 /* 输出链表数据 */
52 /* ----- */
53 void Print_List(Link Head)
54 {
55     Link Pointer;          /* 节点声明 */
56     Pointer = Head;        /* Pointer 指针设为首节点 */
57     While ( Pointer != NULL ) /* 当节点为 NULL 结束循环 */
58     {
59         printf("[%d,%d]", Pointer->Number, Pointer->Total);
60         Pointer = Pointer->Next;    /* 往下一个节点 */
61     }
62     printf("\n");
63 }
64
65 /* ----- */
66 /* 释放链表 */
67 /* ----- */
68 void Free_List(Link Head)
69 {
70     Link Pointer;          /* 节点声明 */
71
72     While ( Head != NULL )    /* 当节点为 NULL 结束循环 */
73     {
74         Pointer = Head;
75         Head = Head->Next;    /* 往下一个节点 */
76         Free(Pointer);
77     }
78 }
79
80 /* ----- */
81 /* 建立链表 */
82 /* ----- */
83 Link Create_List(Link Head)
84 {
85     Link New;              /* 节点声明 */
86     Link Pointer;          /* 节点声明 */
87     int i;
88
89     Head = (Link) malloc(sizeof(Node));    /* 内存配置 */

```

```

90
91     if ( Head == NULL )
92     printf("Memory allocate Failure!!\n"); /* 内存配置失败 */
93     /*
94     else
95     {
96         Head->Number = Data[0][0]; /* 定义首节点数据编号 */
97         Head->Total = Data[1][0];
98         Head->Next = NULL;
99
100        Pointer = Head;      /* Pointer 指针设为首节点 */
101
102        for ( i=1;i<Max;i++)
103        {
104            New = (Link) malloc(sizeof(Node)); /* 内存配置 */
105
106            New->Number = Data[0][i];
107            New->Total = Data[1][i];
108            New->Next = NULL;
109
110            Pointer->Next = New; /* 将新节点串连在原列表尾端 */
111            /*
112            Pointer = New;      /* 列表尾端节点为新节点 */
113        }
114    }
115    return Head;
116 }
117 /* ----- */
118 /* 主程序 */
119 /* ----- */
120 void main ()
121 {
122     Link  Head;      /* 节点声明 */
123     Link  New;
124     int    Key;
125
126     Head = Create_List(Head); /* 调用建立链表 */
127
128     if ( Head != NULL )
129     {
130         Print_List(Head);
131         While ( 1 )
132         {
133             printf("Input 0 to EXIT\n"); /* 数据输入提示 */
134             New = (Link) malloc(sizeof(Node)); /* 内存配置 */
135             printf("Please input the data number : ");
136             scanf("%d",&New->Number);
137             if ( New->Number == 0 ) /* 输入 0 时结束循环 */
138                 break;
139             printf("Please input the data total : ");
140             scanf("%d",&New->Total);
141             printf("Please input the data number for Insert : ");
142             scanf("%d",&Key);
143
144             Head = Insert_List(Head,New,Key); /* 调用插入节点 */
145             /*
146             Print_List(Head); /* 输出链表数据 */
147         }
148         Free_List(Head); /* 调用释放链表 */
149     }
150 }

```

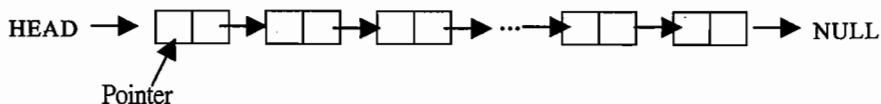

运行结果:

```
C:\DS>insert
[1,15][3,35][5,10][7,67][2,25][4,65][6,38][8,70][9,30][0,20]
Input 0 to EXIT
Please input the data number : 10
Please input the data total : 8
Please input the data number for Insert : 11
[10,8][1,15][3,35][5,10][7,67][2,25][4,65][6,38][8,70][9,30][0,20]
Input 0 to EXIT
Please input the data number : 20
Please input the data total : 6
Please input the data number for Insert : 7
[10,8][1,15][3,35][5,10][7,67][20,6][2,25][4,65][6,38][8,70][9,30][0,20]
Input 0 to EXIT
Please input the data number : 30
Please input the data total : 7
Please input the data number for Insert : 0
[10,8][1,15][3,35][5,10][7,67][20,6][2,25][4,65][6,38][8,70][9,30][0,20][30,7]
Input 0 to EXIT
Please input the data number : 0
C:\DS>
```

3.3.2 单链表内节点的删除

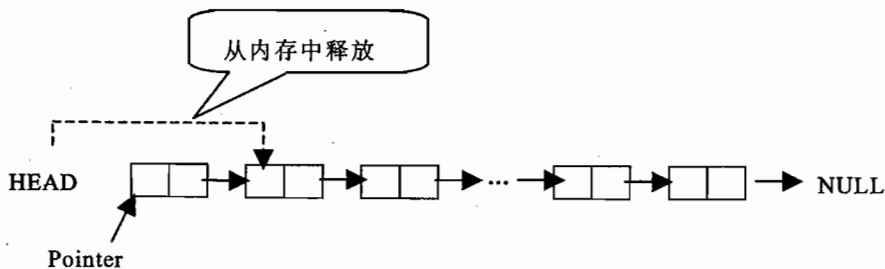
能插入新的节点, 就能删除链表内的节点, 一般而言, 删除节点也可分为下列几类:

1. 删除链表首节点

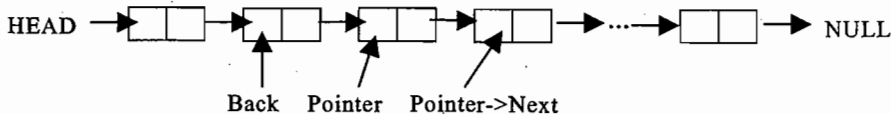


删除的节点在链表的开头需要将首节点指向首节点的指针(即下一个节点), 并将原来的节点从内存中释放。

```
HEAD = Pointer->Next
Free(Pointer)
```

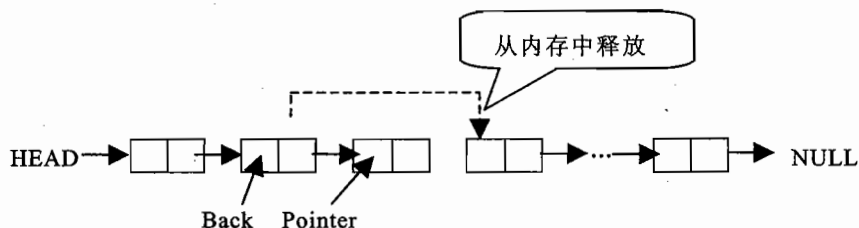


2. 删除链表中间节点

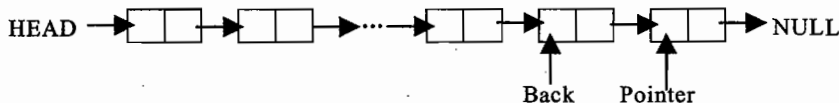


删除的节点在链表的中间, 如果我们找到 Pointer 节点, 则需要将前一个节点的指针指向 Pointer 节点的指针(即下一个节点)。并将原来的节点从内存中释放。

```
Back->Next = Pointer->Next
Free(Pointer)
```

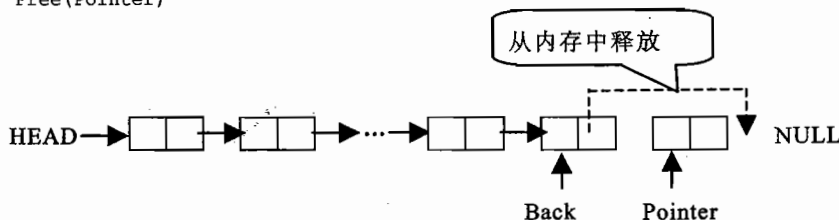


3. 删除链表尾端的节点



删除的节点在链表的尾端(Pointer 节点), 如果我们找到 Pointer 节点, 则需要将前一个节点的指针指向 Pointer 节点的指针(NULL)。并将原来的节点从内存中释放。

```
Back->Next = Pointer->Next
Free(Pointer)
```



程序实例:

设计一个删除链表内节点的程序。

程序构思:

持续往下一个节点查找欲删除节点, 直到节点内容找到或节点指针为 NULL(即找不到该节点)。

在删除时, 必须记录前一个节点的位置(Back)。

如果该节点不存在, 输出消息说节点不存在。

如果该节点存在, 且是首节点:

```
HEAD = Pointer->Next
Free(Pointer)
```

如果该节点存在, 但非首节点(即链连列表中节点或尾端节点), 则:

```
Back->Next = Pointer->Next
Free(Pointer)
```

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: delete.c */
03  /* 程序目的: 设计一个删除链表内节点的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  #define Max 10
08
```

```

09     struct    List                /* 节点结构声明 */
10     {
11         int      Number;
12         int      Total;
13         struct List *Next;
14     };
15     typedef    struct    List Node;
16     typedef    Node *Link;
17
18     int Data[2][Max] = /* 输入数据 */
19         { 1, 3, 5, 7, 2, 4, 6, 8, 9, 10,
20           15, 35, 10, 67, 25, 65, 38, 70, 30, 20 };
21
22     /* ----- */
23     /* 删除链表内节点 */
24     /* ----- */
25     Link Delete_List(Link Head,int Key)
26     {
27         Link  Pointer;          /* 节点声明 */
28         Link  Back;
29
30         Pointer = Head;        /* Pointer 指针设为首节点 */
31
32         while ( 1 )
33         {
34             if ( Pointer->Next == NULL )
35             {
36                 printf("Not Found!!\n");
37                 break;
38             }
39             if ( Head->Number == Key ) /* 删除首节点 */
40             {
41                 Head = Pointer->Next;
42                 free(Pointer);
43                 break;
44             }
45             Back = Pointer;
46             Pointer = Pointer->Next; /* 往下一个节点 */
47             if ( Pointer->Number == Key ) /* 插入在链表中间或尾端 */
48             {
49                 Back->Next = Pointer->Next;
50                 free(Pointer);
51                 break;
52             }
53         }
54         return Head;
55     }
56
57     /* ----- */
58     /* 输出链表数据 */
59     /* ----- */
60     void Print_List(Link Head)
61     {
62         Link  Pointer;          /* 节点声明 */
63         Pointer = Head;        /* Pointer 指针设为首节点 */
64         while ( Pointer != NULL ) /* 当节点为 NULL 结束循环 */
65         {
66             printf("[%d,%d]", Pointer->Number, Pointer->Total);
67             Pointer = Pointer->Next; /* 往下一个节点 */
68         }
69         printf("\n");

```

```

70     }
71
72     /* ----- */
73     /* 释放链表 */
74     /* ----- */
75     void Free_List(Link Head)
76     {
77         Link Pointer;          /* 节点声明 */
78
79         while ( Head != NULL ) /* 当节点为 NULL 结束循环 */
80         {
81             Pointer = Head;
82             Head = Head->Next; /* 往下一个节点 */
83             free(Pointer);
84         }
85     }
86
87     /* ----- */
88     /* 建立链表 */
89     /* ----- */
90     Link Create_List(Link Head)
91     {
92         Link New;              /* 节点声明 */
93         Link Pointer;          /* 节点声明 */
94         int i;
95
96         Head = (Link) malloc(sizeof(Node)); /* 内存配置 */
97
98         if ( Head == NULL )
99             printf("Memory allocate Failure!!\n"); /* 内存配置败 */
100
101         else
102         {
103             Head->Number = Data[0][0]; /* 定义首节点数据编号 */
104             Head->Total = Data[1][0];
105             Head->Next = NULL;
106
107             Pointer = Head; /* Pointer 指针设为首节点 */
108
109             For ( i=1; i<Max; i++)
110             {
111                 New = (Link) malloc(sizeof(Node)); /* 内存配置 */
112
113                 New->Number = Data[0][i];
114                 New->Total = Data[1][i];
115                 New->Next = NULL;
116
117                 Pointer->Next = New; /* 将新节点串连在原列表尾端 */
118
119                 Pointer = New; /* 列表尾端节点为新节点 */
120             }
121         }
122         return Head;
123     }
124     /* ----- */
125     /* 主程序 */
126     /* ----- */
127     void main ()
128     {
129         Link Head;              /* 节点声明 */
130         int Key;

```

```
131
132     Head = Create_List(Head);    /* 调用建立链表 */
133
134     if ( Head != NULL )
135     {
136         Print_List(Head);
137         while ( 1 )
138         {
139             printf("Input 0 to EXIT\n");    /* 数据输入提示 */
140             printf("Please input the data number for Delete : ");
141             scanf("%d",&Key);
142             if ( Key == 0 )    /* 输入 0 时结束循环 */
143                 break;
144
145             Head = Delete_List(Head,Key);    /* 调用插入节点 */
146             Print_List(Head);    /* 输出链表数据 */
147         }
148         Free_List(Head);    /* 调用释放链表 */
149     }
150 }
```

运行结果:

```
C:\DS>delete
[1,15][3,35][5,10][7,67][2,25][4,65][6,38][8,70][9,30][10,20]
Input 0 to EXIT
Please input the data number for Delete : 1
[3,35][5,10][7,67][2,25][4,65][6,38][8,70][9,30][10,20]
Input 0 to EXIT
Please input the data number for Delete : 7
[3,35][5,10][2,25][4,65][6,38][8,70][9,30][10,20]
Input 0 to EXIT
Please input the data number for Delete : 10
[3,35][5,10][2,25][4,65][6,38][8,70][9,30]
Input 0 to EXIT
Please input the data number for Delete : 11
Not Found!!
[3,35][5,10][2,25][4,65][6,38][8,70][9,30]
Input 0 to EXIT
Please input the data number for Delete : 0
C:\DS>
```

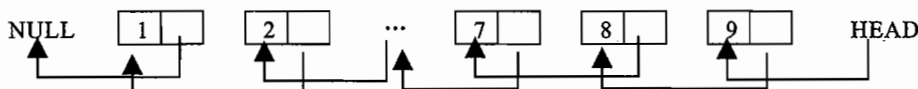
3.3.3 单链表的反转

看了那么多的链表的插入和删除的讨论，现在如果我们想把链表从尾至头反转过来，那么又该如何思考呢？

也就是当原链表为：

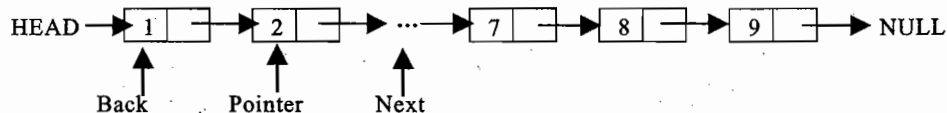


我们想反转为：



其实反转的方式并不如想象中那么困难，其操作方式如下：

步骤 1:



从首节点开始为 Back 节点、Back 节点的下一个节点为 Pointer 节点, 若 Back 节点是首节点, 则将 Back 节点的指针设为 NULL。

```

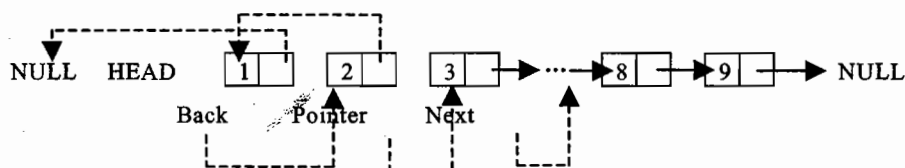
Back = HEAD
Pointer = Back->Next
Back->Next = NULL
  
```

下一个节点(Next 节点)设为 Pointer 节点的指针, 将 Pointer 节点的指针指向上一个节点(Back 节点), Back 节点设为 Pointer 节点, Pointer 节点设为下一个节点(Next 节点)。

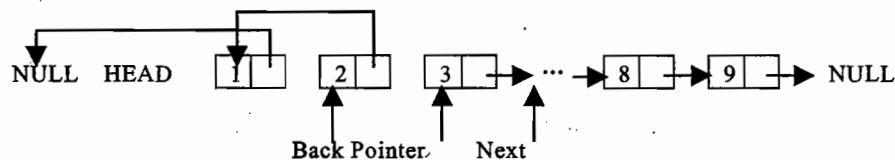
```

Next = Pointer->Next
Pointer->Next = Back
Back = Pointer
Pointer = Next
  
```

如下图所示:



步骤 2:

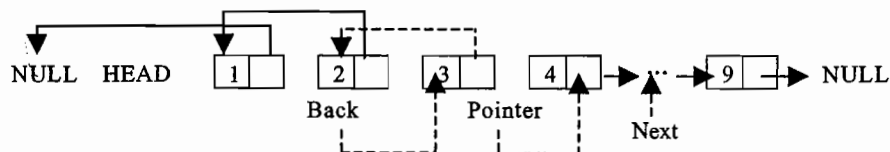


下一个节点设为 Pointer 节点指针, 将 Pointer 节点的指针指向上一个节点, 上一个节点设为 Pointer 节点, Pointer 节点设为下一个节点。

```

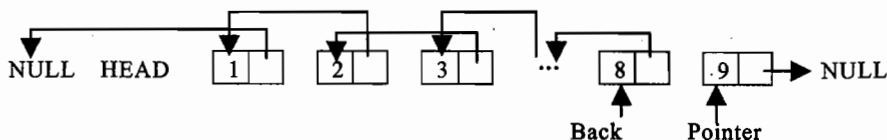
Next = Pointer->Next
Pointer->Next = Back
Back = Pointer
Pointer = Next
  
```

如下图所示:



重复步骤 2, 直到 Pointer 节点的指针指向 NULL 为止。

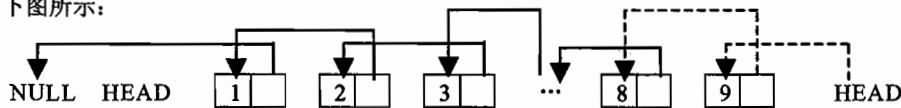
步骤 3:



将 Pointer 节点的指针指向上一个节点, 首节点 HEAD 节点设为 Pointer 节点。

```
Pointer->Next = back
Head = Pointer
```

如下图所示:



程序实例:

设计一个反转链表的程序。

程序构思:

步骤 1: 从首节点开始为节点、下一个节点的下一个节点为 Pointer 节点, 若 Back 节点是首节点, 则将节点的指针设为 NULL。下一个节点设为 Pointer 节点的指针, 将 Pointer 节点的指针指向上一个节点, Back 节点设为 Pointer 节点, Pointer 节点设为下一个节点。

```
Back = HEAD
Pointer = Back->Next
Next = Pointer->Next
Pointer->Next = Back
Back = Pointer
Pointer = Next
```

步骤 2: 下一个节点设为 Pointer 节点指针, 将 Pointer 节点的指针指向上一个节点, 下一个节点设为 Pointer 节点, Pointer 节点设为下一个节点。

```
Next = Pointer->Next
Pointer->Next = Back
Back = Pointer
Pointer = Next
```

重复步骤 2, 直到 Pointer 节点的指针指向 NULL 为止。

步骤 3: 将 Pointer 节点的指针指向上一个节点, 首节点 HEAD 节点设为 Pointer 节点。

```
Pointer->Next = back
Head = Pointer
```

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: invert.c */
03  /* 程序目的: 设计一个反转链表的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  #define Max 10
08
```

```
09 struct List /* 节点结构声明 */
10 {
11     int Number;
12     int Total;
13     struct List *Next;
14 };
15 typedef struct List Node;
16 typedef Node *Link;
17
18 int Data[2][Max] = /* 输入数据 */
19 { 1, 3, 5, 7, 2, 4, 6, 8, 9, 10,
20   15, 35, 10, 67, 25, 65, 38, 70, 30, 20 };
21
22 /* ----- */
23 /* 反转链表 */
24 /* ----- */
25 Link Invert_List(Link Head)
26 {
27     Link Pointer; /* 节点声明 */
28     Link Back; /* 上一个节点 */
29     Link Next; /* 下一个节点 */
30
31     Back = Head; /* Back 指针设为首节点 */
32     Pointer = Back->Next;
33     Back->Next = NULL;
34
35     Next = Pointer->Next;
36     Pointer->Next = Back;
37     Back = Pointer;
38     Pointer = Next;
39
40     while ( Pointer->Next != NULL ) /* 当到达链表尾端时, 结束循环 */
41     {
42         Next = Pointer->Next;
43         Pointer->Next = Back;
44         Back = Pointer;
45         Pointer = Next;
46     }
47     Pointer->Next = Back;
48     Head = Pointer;
49     return Head;
50 }
51
52 /* ----- */
53 /* 输出链表数据 */
54 /* ----- */
55 void Print_List(Link Head)
56 {
57     Link Pointer; /* 节点声明 */
58     Pointer = Head; /* Pointer 指针设为首节点 */
59     while ( Pointer != NULL ) /* 当节点为 NULL 结束循环 */
60     {
61         printf("[%d,%d]", Pointer->Number, Pointer->Total);
62         Pointer = Pointer->Next; /* 往下一个节点 */
63     }
64     printf("\n");
65 }
66
67 /* ----- */
68 /* 释放链表 */
69 /* ----- */
```



```

70 void Free_List(Link Head)
71 {
72     Link Pointer;          /* 节点声明 */
73
74     while ( Head != NULL ) /* 当节点为 NULL 结束循环 */
75     {
76         Pointer = Head;
77         Head = Head->Next; /* 往下一个节点 */
78         free(Pointer);
79     }
80 }
81
82 /* ----- */
83 /* 建立链表 */
84 /* ----- */
85 Link Create_List(Link Head)
86 {
87     Link New;              /* 节点声明 */
88     Link Pointer;         /* 节点声明 */
89     int i;
90
91     Head = (Link) malloc(sizeof(Node)); /* 内存配置 */
92
93     if ( Head == NULL )
94         printf("Memory allocate Failure!!\n"); /* 内存配置失败 */
95     else
96     {
97         Head->Number = Data[0][0]; /* 定义首节点数据编号 */
98         Head->Total = Data[1][0];
99         Head->Next = NULL;
100
101         Pointer = Head; /* Pointer 指针设为首节点 */
102
103         for ( i=1; i<Max; i++)
104         {
105             New = (Link) malloc(sizeof(Node)); /* 内存配置 */
106
107             New->Number = Data[0][i];
108             New->Total = Data[1][i];
109             New->Next = NULL;
110
111             Pointer->Next = New; /* 将新节点串连在原列表尾端 */
112             /*
113             Pointer = New; /* 列表尾端节点为新节点 */
114         }
115     }
116     return Head;
117 }
118 /* ----- */
119 /* 主程序 */
120 /* ----- */
121 void main ()
122 {
123     Link Head; /* 节点声明 */
124
125     Head = Create_List(Head); /* 调用建立链表 */
126
127     if ( Head != NULL )
128     {
129         printf("Input Data : \n");
130         Print_List(Head); /* 调用输出链表数据 */

```

```

131
132     Head = Invert_List(Head); /* 调用反转链表 */
133     printf("After Invert : \n");
134     Print_List(Head);         /* 调用输出链表数据 */
135     Free_List(Head);          /* 调用释放链表 */
136 }
137 }

```

运行结果:

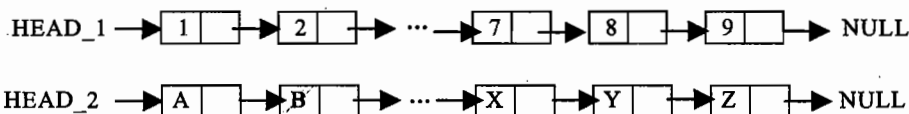
```

C:\DS>invert
Input Data :
[1,15][3,35][5,10][7,67][2,25][4,65][6,38][8,70][9,30][10,20]
After Invert :
[10,20][9,30][8,70][6,38][4,65][2,25][7,67][5,10][3,35][1,15]
C:\DS>

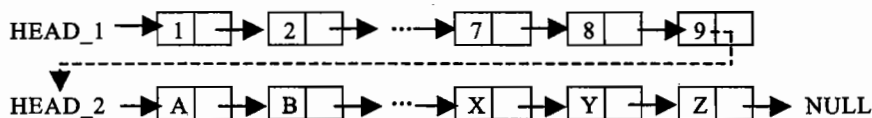
```

3.3.4 单链表的链接

有两个链表如下图所示:



两个链表的链接,是将第2个链表的首节点,串结在第1个链表的尾端。两个链表的链接之后如下:



程序实例:

设计一个将两个链表链接的程序。

程序构思:

找到第1个链表的尾端,将该节点指针指向第2个节点的首节点。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称:  concate.c                      */
03  /* 程序目的:  设计一个将两个链表链接的程序。      */
04  /* Written By Kuo-Yu Huang. (WANT Studio.)        */
05  /* ===== */
06  #include <stdlib.h>
07  #define Max1 10
08  #define Max2 6
09
10  struct List          /* 节点结构声明 */
11  {
12      int      Number;
13      struct List *Next;
14  };

```

```

15     typedef struct List Node;
16     typedef Node *Link;
17
18     int Data1[Max1] = /* 输入数据 */
19         { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
20
21     int Data2[Max2] = /* 输入数据 */
22         { 11, 12, 13, 14, 15, 16 };
23
24     /* ----- */
25     /* 链接链表 */
26     /* ----- */
27     Link Concatenate_List(Link Head1, Link Head2)
28     {
29         Link Pointer; /* 节点声明 */
30         Link Back; /* 上一个节点 */
31         Link Next; /* 下一个节点 */
32
33         Pointer = Head1; /* Pointer 指针设为首节点 */
34
35         /* 当到达链表尾端时, 结束循环 */
36         while ( Pointer->Next != NULL )
37             Pointer = Pointer->Next;
38
39         Pointer->Next = Head2;
40
41         return Head1;
42     }
43
44     /* ----- */
45     /* 输出链表数据 */
46     /* ----- */
47     void Print_List(Link Head)
48     {
49         Link Pointer; /* 节点声明 */
50         Pointer = Head; /* Pointer 指针设为首节点 */
51         while ( Pointer != NULL ) /* 当节点为 NULL 结束循环 */
52         {
53             printf("[%d]", Pointer->Number);
54             Pointer = Pointer->Next; /* 往下一个节点 */
55         }
56         printf("\n");
57     }
58
59     /* ----- */
60     /* 释放链表 */
61     /* ----- */
62     void Free_List(Link Head)
63     {
64         Link Pointer; /* 节点声明 */
65
66         while ( Head != NULL ) /* 当节点为 NULL 结束循环 */
67         {
68             Pointer = Head;
69             Head = Head->Next; /* 往下一个节点 */
70             free(Pointer);
71         }
72     }
73
74     /* ----- */
75     /* 建立链表 */

```

```

76  /* ----- */
77  Link Create_List(Link Head,int *Data,int Max)
78  {
79      Link New;          /* 节点声明 */
80      Link Pointer;      /* 节点声明 */
81      int i;
82
83      Head = (Link) malloc(sizeof(Node));          /* 内存配置 */
84
85      if ( Head == NULL )
86          printf("Memory allocate Failure!!\n");    /* 内存配置失败 */
87      /*
88      else
89      {
90          Head->Number = Data[0];    /* 定义首节点数据编号 */
91          Head->Next = NULL;
92
93          Pointer = Head;          /* Pointer 指针设为首节点 */
94
95          for ( i=1;i<Max;i++)
96          {
97              New = (Link) malloc(sizeof(Node)); /* 内存配置 */
98
99              New->Number = Data[i];
100             New->Next = NULL;
101
102             Pointer->Next = New; /* 将新节点串连在原列表尾端 */
103             /*
104             Pointer = New;          /* 列表尾端节点为新节点 */
105         }
106     }
107     return Head;
108 }
109 /* ----- */
110 /* 主程序 */
111 /* ----- */
112 void main ()
113 {
114     Link Head;
115     Link Head1;          /* 节点声明 */
116     Link Head2;          /* 节点声明 */
117
118     Head1 = Create_List(Head1,Data1,Max1); /* 调用建立链表 */
119     /*
120     Head2 = Create_List(Head2,Data2,Max2); /* 调用建立链表 */
121     /*
122
123     if ( Head1 != NULL && Head2 != NULL )
124     {
125         printf("Input Data : \n");
126         Print_List(Head1);          /* 调用输出链表数据 */
127         Print_List(Head2);
128
129         Head = Concatenate_List(Head1,Head2); /* 调用反转链表 */
130         /*
131         printf("After Concatenate : \n");
132         Print_List(Head);          /* 调用输出链表数据 */
133         Free_List(Head);          /* 调用释放链表 */
134     }
135 }

```

运行结果:

```
C:\DS>concat
Input Data :
[1][2][3][4][5][6][7][8][9][10]
[11][12][13][14][15][16]
After Concatenate :
[1][2][3][4][5][6][7][8][9][10][11][12][13][14][15][16]
C:\DS>
```

3.3.5 单链表的比较

链表的比较是用于判断两个链表内的数据是否相等。

程序实例:

设计一个比较两个链表内容的程序。

程序构思:

找到第1个链表的首节点与第2个链表的首节点,依次比较其节点内容,直到某一个链表到达尾端为止。

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: compare.c */
03  /* 程序目的: 设计一个将两个链表链接的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  #define Max1 10
08  #define Max2 10
09  #define Max3 6
10
11  struct List /* 节点结构声明 */
12  {
13      int Number;
14      struct List *Next;
15  };
16  typedef struct List Node;
17  typedef Node *Link;
18
19  int Data1[Max1] = /* 输入数据1 */
20      { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
21
22  int Data2[Max2] = /* 输入数据2 */
23      { 1, 2, 5, 4, 3, 7, 6, 10, 9, 8 };
24
25  int Data3[Max3] = /* 输入数据3 */
26      { 1, 2, 3, 4, 5, 6 };
27
28  /* ----- */
29  /* 比较链表内容 */
30  /* ----- */
31  void Compare_List(Link Head1, Link Head2)
32  {
```

```

33     Link  Pointer1;      /* 节点声明 */
34     Link  Pointer2;      /* 节点声明 */
35
36     Pointer1 = Head1; /* Pointer1 指针设为第一个链表首节点
37     */
38     Pointer2 = Head2; /* Pointer1 指针设为第一个链表首节点
39     */
40
41     while ( 1 )
42     {
43         /* 当两链表指针都达尾端时, 表示相等 */
44         if ( ( Pointer1->Next == NULL ) && ( Pointer2->Next == NULL ) )
45         {
46             printf("\n##EQUAL!!##\n");
47             break;
48         }
49         /* 当两链表节点内容不同时, 表示不相等 */
50         if ( Pointer1->Number != Pointer2->Number )
51         {
52             printf("\n[%d] <>
53 [%d]\n", Pointer1->Number, Pointer2->Number);
54             printf("##NOT EQUAL!!##\n");
55             break;
56         }
57         else /* 节点内容相等时, 往下一个节点 */
58         {
59             printf("[%d]", Pointer1->Number);
60             Pointer1 = Pointer1->Next;
61             Pointer2 = Pointer2->Next;
62         }
63     }
64 }
65
66 /* ----- */
67 /* 输出链表数据 */
68 /* ----- */
69 void Print_List(Link Head)
70 {
71     Link  Pointer;      /* 节点声明 */
72     Pointer = Head;      /* Pointer 指针设为首节点 */
73     while ( Pointer != NULL ) /* 当节点为 NULL 结束循环 */
74     {
75         printf("[%d]", Pointer->Number);
76         Pointer = Pointer->Next; /* 往下一个节点 */
77     }
78     printf("\n");
79 }
80
81 /* ----- */
82 /* 释放链表 */
83 /* ----- */
84 void Free_List(Link Head)
85 {
86     Link  Pointer;      /* 节点声明 */
87
88     while ( Head != NULL ) /* 当节点为 NULL 结束循环 */
89     {
90         Pointer = Head;
91         Head = Head->Next; /* 往下一个节点 */
92         free(Pointer);
93     }

```

```

94     }
95
96     /* ----- */
97     /* 建立链表 */
98     /* ----- */
99     Link Create_List(Link Head,int *Data,int Max)
100    {
101        Link    New;           /* 节点声明 */
102        Link    Pointer;       /* 节点声明 */
103        int     i;
104
105        Head = (Link) malloc(sizeof(Node));    /* 内存配置 */
106
107        if ( Head == NULL )
108            printf("Memory allocate Failure!!\n");    /* 内存配置失败 */
109        /*
110        else
111        {
112            Head->Number = Data[0];    /* 定义首节点数据编号 */
113            Head->Next = NULL;
114
115            Pointer = Head;    /* Pointer 指针设为首节点 */
116
117            for ( i=1;i<Max;i++)
118            {
119                New = (Link) malloc(sizeof(Node)); /* 内存配置 */
120
121                New->Number = Data[i];
122                New->Next = NULL;
123
124                Pointer->Next = New; /* 将新节点串连在原列表尾端 */
125                /*
126                Pointer = New;    /* 列表尾端节点为新节点 */
127            }
128        }
129        return Head;
130    }
131    /* ----- */
132    /* 主程序 */
133    /* ----- */
134    void main ()
135    {
136        Link    Head1;           /* 节点声明 */
137        Link    Head2;           /* 节点声明 */
138        Link    Head3;           /* 节点声明 */
139        int     Flag;
140
141        Head1 = Create_List(Head1,Data1,Max1); /* 调用建立链表 */
142        /*
143        Head2 = Create_List(Head2,Data2,Max2); /* 调用建立链表 */
144        /*
145        Head3 = Create_List(Head3,Data3,Max3); /* 调用建立链表 */
146        /*
147
148        if ( Head1 != NULL && Head2 != NULL && Head3 != NULL)
149        {
150            printf("Linked List 1 : ");
151            Print_List(Head1);    /* 调用输出链表数据 */
152            printf("Linked List 2 : ");
153            Print_List(Head2);    /* 调用输出链表数据 */
154            printf("Linked List 3 : ");

```

```

155      Print_List(Head3);          /* 调用输出链表数据 */
156
157      printf("Linked List 1 compare with Linked List 1\n");
158      Compare_List(Head1,Head1);/* 调用比较链表 */
159
160      printf("Linked List 1 compare with Linked List 2\n");
161      Compare_List(Head1,Head2);/* 调用比较链表 */
162
163      printf("Linked List 1 compare with Linked List 3\n");
164      Compare_List(Head1,Head3);/* 调用比较链表 */
165
166      Free_List(Head1);            /* 调用释放链表 1 */
167      Free_List(Head2);            /* 调用释放链表 2 */
168      Free_List(Head3);            /* 调用释放链表 3 */
169  }
170  }

```

运行结果:

```

C:\DS>compare
Linked List 1 : [1][2][3][4][5][6][7][8][9][10]
Linked List 2 : [1][2][5][4][3][7][6][10][9][8]
Linked List 3 : [1][2][3][4][5][6]
Linked List 1 compare with Linked List 1
[1][2][3][4][5][6][7][8][9]
##EQUAL!!##
Linked List 1 compare with Linked List 2
[1][2]
[3] <> [5]
##NOT EQUAL!!##
Linked List 1 compare with Linked List 3
[1][2][3][4][5][6]
[7] <> [0]
##NOT EQUAL!!##
C:\DS>

```

【习题】

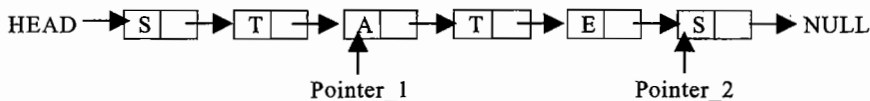
一、复习:

1. 单链表是由多个节点所串连而成的, 每一个节点包含了数据部分和指向链表的下一个节点的指针。
2. 下列那一个程序片段是在链表中间插入一个节点。(假设新节点为 NEW, 欲插入在 Pointer 节点之后)
 - (a) NEW->Next = Pointer
Pointer = NEW
 - (b) NEW->Next = Pointer->Next
Pointer->Next = NEW
 - (c) Pointer->Next = NEW->Next
NEW->Next = Pointer
 - (d) 以上皆非。

3. 下列那一个程序片段是删除链表中间节点。(假设欲删除节点为 Pointer 节点、Back 为前一个节点。)

- (a) `free(Pointer);`
- (b) `free(Back);`
- (c) `Back = Pointer->Next;`
`free(Pointer);`
- (d) `Back->Next = Pointer->Next;`
`free(Pointer);`

4. 如有一个链表如下:



下列语句哪一个正确。

```

HEAD->Next->Data == "A"
HEAD->Data == "T"
Pointer_1->Data == "A"
Pointer_2->Next == NULL
  
```

二、应用:

试利用单链表编写一个学生成绩系统。(具有查询成绩、修改成绩、删除成绩、添加成绩、全班平均。)

数据如下:

学生座号	学生姓名	语文成绩	英语成绩	数学成绩
6	Alan	85	90	98
15	Danie	76	70	80
17	Helen	95	98	96
20	Bill	65	60	80
23	Peter	79	65	86
32	Amy	93	86	74



堆 栈

第 4 章

- ◆ 何谓堆栈
- ◆ 用数组仿真堆栈
- ◆ 用链表仿真堆栈
- ◆ 表达式表示法
- ◆ 中序表达式的计算
- ◆ 前序表达式的计算
- ◆ 后序表达式的计算
- ◆ 表达式的转换

“堆栈”(Stack)和“队列”(Queue)是在程序设计时经常使用的数据结构,两者都是有序列表(ordered list),只是对于其内部数据之存取方式不同。本章将先介绍“堆栈”的结构、实作及应用,下一章将针对“队列”做详细的说明。

4.1 何谓堆栈

堆栈数据结构的特性是只允许数据自有序列表的一个固定端(前端)做输入、输出动作,如此一来会使得最后被输入的数据会最先被取出来,也就是具有先进后出 FILO(First In Last Out)的特性。

堆栈的应用相当多,下面列出几项较常见的:

1. 子程序之调用:

在跳往子程序前,会先将下一个指令的地址存到堆栈中,直到子程序执行完后再将地址取出,再回到原来的程序中。

2. 处理递归调用:

和子程序的调用类似,只是除了存储下一个指令的地址外,也将参数、区域变量等数据存入堆栈中。

3. 表达式之转换与求值。

4. 二叉树的遍历。

5. 图形的深度优先(depth-first)追踪法。

建立堆栈可使用的两种结构:数组结构和链表结构,将分别在 4.2 节及 4.3 节做介绍。

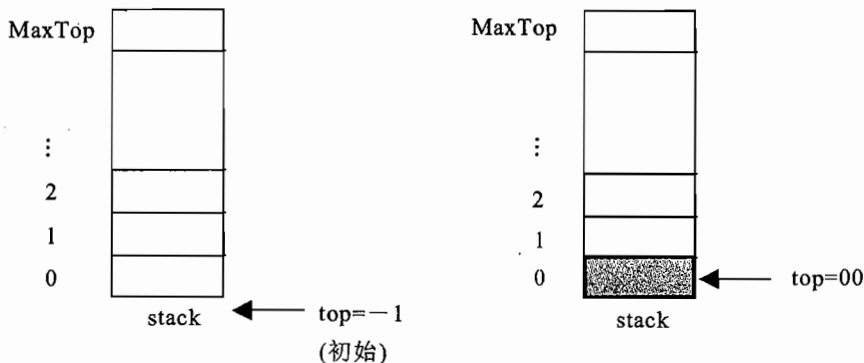
4.2 用数组仿真堆栈

由于堆栈是一种有序列表,当然可以使用数组的结构来存储堆栈的数据内容,堆栈数组的声明如下:

```
char stack[MaxSize];  
int top=-1;
```

其中 MaxSize 是该堆栈的最大容量。后面将会以 MaxSize-1 的值 MaxTop 作为堆栈最大顶端指针。虽然数组结构可直接用索引来存取数据,但在此将数组视为堆栈,故只能从堆栈的顶端进行处理。所以需要有一个变量来记录当前堆栈顶端的索引值。初始值设-1 表示堆栈为空, top 会随着堆栈中数据量的异动改变其指向顶端的位置。

堆栈的数组结构如下图所示:

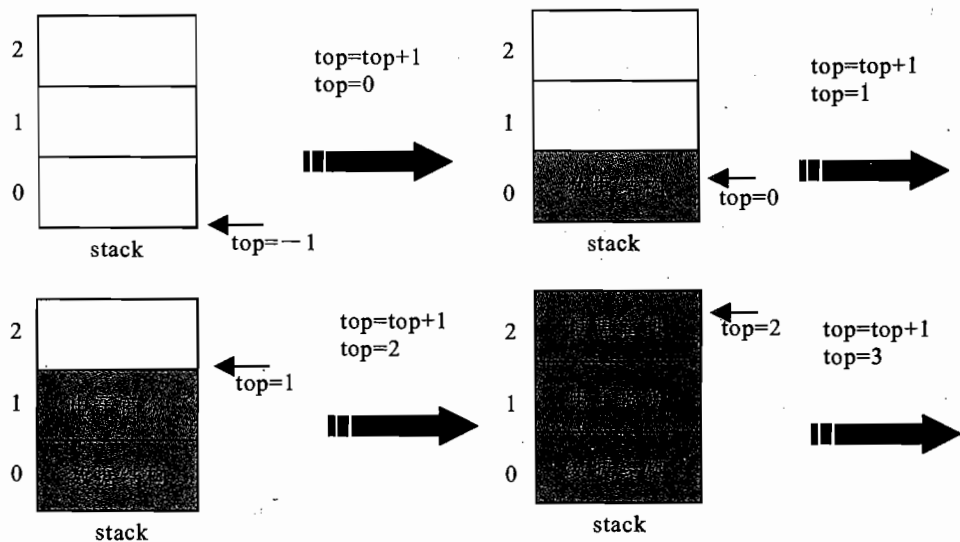


接下来说明在堆栈中是如何对数据做存取。将数据输入堆栈中称为“push”, push 处理主要有两步:

(1) 堆栈顶端指针: $\text{top} + 1$

(2) 若 top 小于等于堆栈最大顶端指针 MaxTop (表示堆栈未滿), 则将数据存入 top 所指的数组元素中, 否则即表示堆栈已滿, 无法存入数据。

例如欲以堆栈的方式将4本书置入箱子($\text{MaxTop}=2$)中, 依序为“数据结构”、“电子商务”、“信息管理”及“投资理财”。Push 的处理如下:



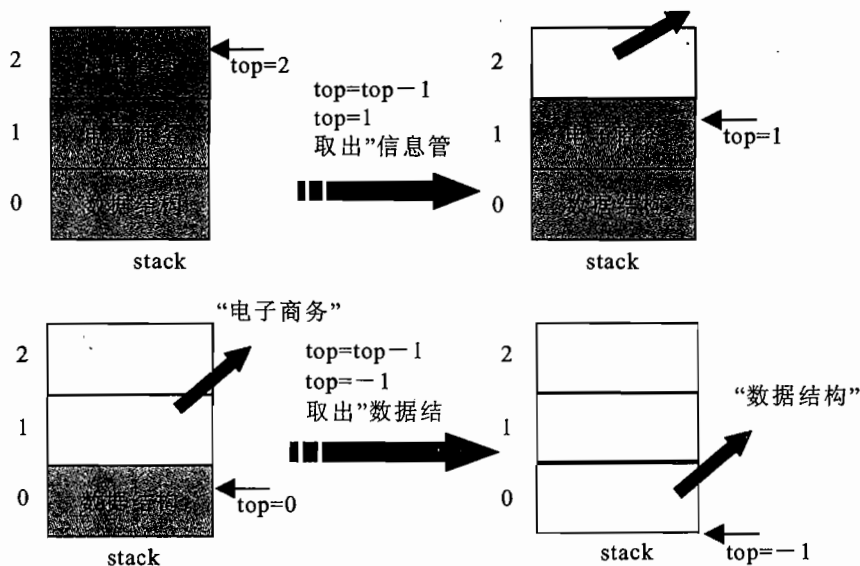
此时堆栈的顶端指针 $\text{top}=3$ 已经大于堆栈最大指针 $\text{MaxTop}=2$, 故无法继续置入第4本“投资理财”。

另外, 将数据从堆栈中取出数据项称为“pop”。pop 的处理也有二个步骤:

(1) 若堆栈指针索引大于等于0时(堆栈未空), 则取出目前堆栈顶端指针 top 所指的数组内容。

(2) 将堆栈顶端指针: $\text{top} - 1$, 指向下一个堆栈元素。

例如欲将箱内的3本书取出, pop 的处理如下:



此时 $\text{top}=-1$ 已小于0, 表示堆栈已空, 无法再从堆栈中取出书本。

在下面的程序中包含堆栈数据的输出及输入处理，并打印目前堆栈的状况。

程序实例：

堆栈数据的存取(使用数组)。

程序构思：

数据输入时，控制堆栈指针+1，若堆栈未满，则将数据存入堆栈中；数据输出时，若堆栈未空，则从堆栈中取出数据，并控制堆栈指针-1。

程序源代码：

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Stack_01.c */
03  /* 程序目的: 堆栈数据之存取(使用数组) */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06
07  #include <stdlib.h>
08  #define MaxSize 10
09  int stack[MaxSize]; /*声明堆栈数组*/
10  int top=-1; /*堆栈顶端*/
11
12  /*-----*/
13  /*      存入堆栈数据      */
14  /*-----*/
15  void push(int value)
16  {
17      int i;
18
19      if (top >= MaxSize) /*判断是否已超出堆栈最大容量*/
20          printf("\nThe stack is full!!\n");
21      else
22      {
23          printf("\nThe stack content before(top->bottom):");
24          for (i=top;i>=0;i--)
25              printf("[%d]",stack[i]);
26
27          top++; /*堆栈顶端指针+1*/
28          stack[top]=value; /*将数据存入堆栈中*/
29
30          printf("\nThe stack content after push(top->bottom):");
31          for (i=top;i>=0;i--)
32              printf("[%d]",stack[i]);
33          printf("\n");
34      }
35  }
36  /*-----*/
37  /*      从堆栈中取出数据      */
38  /*-----*/
39  int pop( )
40  {
41      int temp; /*暂存从堆栈取出来的数据*/
42      int i;
43
44      if (top < 0) /*判断堆栈是否为空*/
45      {

```

```

46         printf("\nThe stack is empty!!\n");
47         return -1;
48     }
49     printf("\nThe stack content before(top->bottom):");
50     for (i=top;i>=0;i--)
51         printf("[%d]",stack[i]);
52
53     temp=stack[top];          /*将取出数据暂存于变量中*/
54     top--;                    /*堆栈顶端指针-1*/
55     printf("\nThe pop value is [%d] ",temp);
56     printf("\nThe stack content after pop(top->bottom):");
57     for (i=top;i>=0;i--)
58         printf("[%d]",stack[i]);
59     printf("\n");
60     return temp;
61 }
62 /*-----*/
63 /*主程序:可输入输出堆栈数据,并输出堆栈内容          */
64 /*-----*/
65 void main ( )
66 {
67     int select;      /*选择功能变量*/
68     int stack[5];    /*堆栈数组*/
69     int i,value;
70
71     printf("\n(1)Input a stack data");
72     printf("\n(2)Output a stack data");
73     printf("\n(3)Exit");
74     printf("\nPlease select one=>");
75     scanf("%d",&select);
76     do
77     {
78         switch (select) /*判断选择的功能*/
79         {
80             case 1: printf("\nPlease input the data=>");
81                     scanf("%d",&value);
82                     push(value); /*将Value存入堆栈*/
83                     break;
84             case 2: value=pop( ); /*从堆栈取出一元素存入Value*/
85                     break;
86         }
87         printf("\n(1)Input a stack data");
88         printf("\n(2)Output a stack data");
89         printf("\n(3)Exit");
90         printf("\nPlease select one=>");
91         scanf("%d",&select);
92         printf("\n");
93     } while (select !=3);
94 }

```

运行结果:

```

C:\DS>Stack_01
(1)Input a stack data
(2)Output a stack data
(3)Exit
Please select one=>1
Please input the data=>4

The stack content before(top->bottom): [3] [2] [1]

```

```
The stack content after push(top->bottom): [4] [3] [2] [1]
```

```
(1)Input a stack data  
(2)Output a stack data  
(3)Exit  
Please select one=>2
```

```
The stack content before(top->bottom): [4] [3] [2] [1]  
The pop value is [4]  
The stack content after pop(top->bottom): [3] [2] [1]
```

```
C:\DS>
```

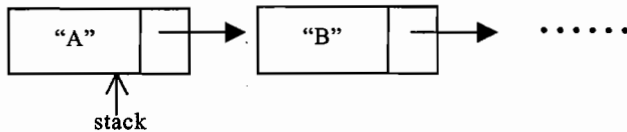
4.3 用链表仿真堆栈

有序列表的表示法除了数组结构外，另外一种就是链表结构，堆栈链表结构的声明如下：

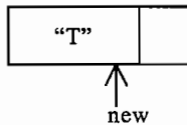
```
struct s_node  
{ int data;  
  struct s_node *point;  
}  
typedef struct s_node s_list;  
typedef s_list *linklist;  
linklist stack=NULL;
```

其中堆栈开始的指针“stack”即为指向堆栈链表顶端的指针。由于堆栈的初始为空，故在声明时先将“stack”设为“NULL”。接下来要说明在堆栈链表中是如何对数据做存取。

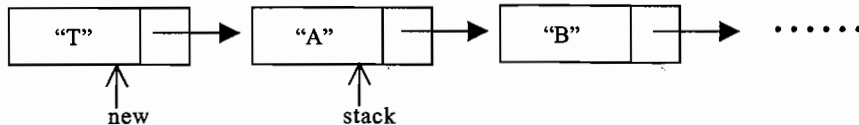
假设已知一堆栈链表如下，欲插入一新数据项“T”于 stack，函数 push() 的处理步骤如下：



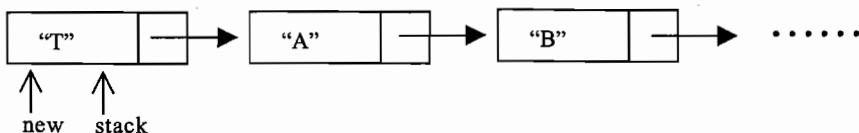
步骤 1：建立一个新节点后存入新数据项内容“T”。



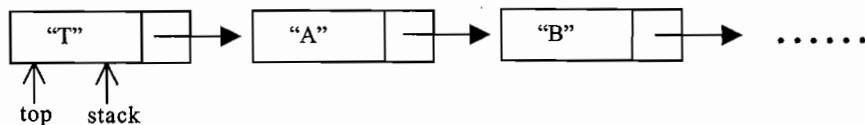
步骤 2：将新节点的指针指向原来堆栈指针所指的节点。



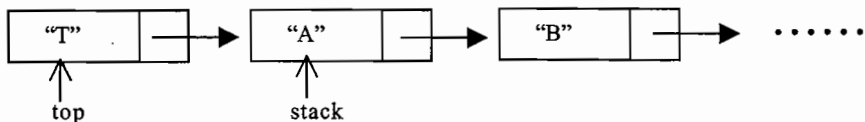
步骤 3：将原堆栈指针指向新节点。



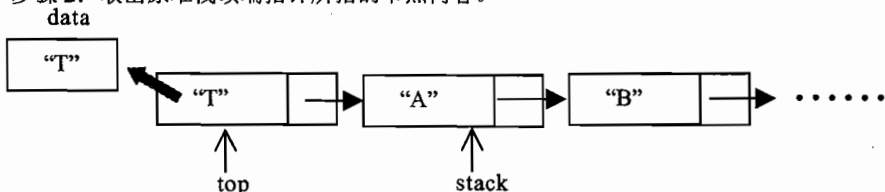
由于堆栈的输出、输入都是从堆栈的顶端进行，故链表堆栈数据的输出如下：



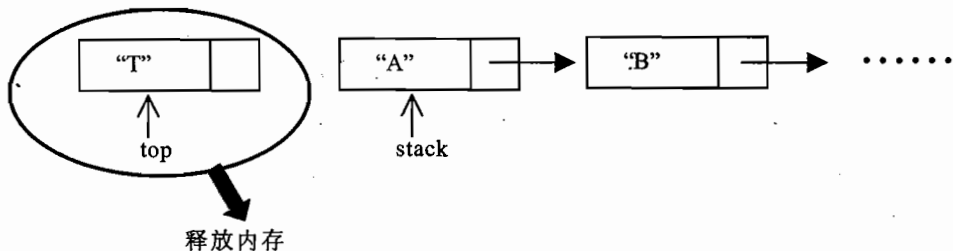
步骤 1：将堆栈指针指向下一个节点。



步骤 2：取出原堆栈顶端指针所指的节点内容。



步骤 3：释放原堆栈顶端指针所指的节点内存。



程序实例：

堆栈数据的存取（使用链表）

程序构思：

数据输入时，控制堆栈尾指针 rear 往前移，若堆栈未满，则将数据存入堆栈中；数据输出时，若堆栈未空，则从堆栈中取出数据，并控制堆栈头指针往前移，并释放取出节点的空间。

程序源代码：

```

01  /* ===== Program Description ===== */
02  /* 程序名称： Stack_02.c */
03  /* 程序目的：堆栈数据的存取（使用链表） */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  struct s_node /*声明堆栈链接结构*/
08  {
09      int data; /*堆栈数据*/
10      struct s_node *next; /*链接指针*/
11  };
12  typedef struct s_node s_list; /*定义新类型列表*/
13  typedef s_list *link; /*定义新类型列表指针*/
  
```



```

14
15     link stack=NULL;
16     /*----- */
17     /* 打印堆栈内容 */
18     /*----- */
19     void print_stack( )
20     {
21         link temp=NULL;
22
23         temp=stack;
24         if (temp==NULL)          /*判断堆栈是否为空*/
25             printf("The stack is empty!!\n");
26         else
27         {
28             while (temp!=NULL)
29             {
30                 printf("[%d] ",temp->data);
31                 temp=temp->next;
32             }
33             printf("\n");
34         }
35     }
36     /*----- */
37     /*      存入堆栈数据 */
38     /*----- */
39     void push(int value)
40     {
41         link newnode; /*声明新节点指针*/
42
43         printf("\nThe stack content before(top->bottom):");
44         print_stack( );
45
46         /*配置节点内存*/
47         newnode=(link) malloc (sizeof(s_list));
48         newnode->data=value;    /*建立新节点*/
49         newnode->next=stack;    /*将新节点指向原堆栈*/
50         stack=newnode;         /*新节点为堆栈之顶端*/
51     }
52     /*----- */
53     /*      从堆栈中取出数据 */
54     /*----- */
55     int pop( )
56     {
57         link top;    /*堆栈的顶端*/
58         int temp;
59
60         printf("\nThe stack content before(top->bottom):");
61         print_stack( );
62         if (stack !=NULL)
63         {
64             top=stack;          /*指向堆栈的顶端*/
65             stack=stack->next;   /*指向下一个节点*/
66             temp=top->data;      /*取出顶点数据*/
67             free(top);          /*释放节点空间*/
68             return temp;        /*返回堆栈指针*/
69         }
70         else
71             return -1;
72     }
73     /*----- */
74     /*      主程序:可输入输出堆栈数据,并输出堆栈内容 */
75     /*----- */

```

```

75 void main ( )
76 {
77     link point;
78     int select;      /*选择功能变量*/
79     int i,value;
80
81     printf("\n(1)Input a stack data");
82     printf("\n(2)Output a stack data");
83     printf("\n(3)Exit");
84     printf("\nPlease select one function=>");
85     scanf("%d",&select);
86     do
87     {
88         switch (select) /*判断选择的功能*/
89         {
90             case 1: printf("\nPlease input the data=>");
91                     scanf("%d",&value);
92                     push(value);      /*将Value存入堆栈*/
93                     printf("The stack content current(top->bottom):");
94                     print_stack( );    /*打印堆栈内容值*/
95                     break;
96             case 2: value=pop( );      /*从堆栈取出一元素存入Value*/
97                     printf("The output value is [%d]",value);
98                     printf("\n");
99                     printf("The stack content current(top->bottom):");
100                    print_stack( );    /*打印堆栈内容值*/
101                    break;
102         }
103         printf("\n(1)Input a stack data");
104         printf("\n(2)Output a stack data");
105         printf("\n(3)Exit");
106         printf("\nPlease select one=>");
107         scanf("%d",&select);
108     } while (select !=3);
109 }

```

运行结果:

```

C:\DS>Stack_02
(1)Input a stack data
(2)Output a stack data
(3)Exit
Please select one function=>1
Please input the data=>1

The stack content before(top->bottom):The stack is empty!!
The stack content current(top->bottom):[1]

(1)Input a stack data
(2)Output a stack data
(3)Exit
Please select one=>2

The stack content before(top->bottom): [1]
The output value is [1]
The stack content current(top->bottom): The stack is empty!!

(1)Input a stack data
(2)Output a stack data
(3)Exit

```

```
Please select one=>3
```

```
C:\DS>
```

4.4 表达式表示法

一个表达式是由操作数(operand)、运算符(operator)及分隔符(delimiter)所构成,一般我们所写的都是中序表示法(infix),也就是将运算符写在两个操作数之间,例如:

$$X+Y$$

$$X+Y*Z$$

由于表达式中的运算符有优先级,故要用计算机运算中序表达式是很不方便的。所以要在计算机中有效率地计算表达式的值,我们希望能将中序表达式先转换成较易求值的前序(prefix)或后序(postfix)表达式,再进而求算式之值。

前序、中序及后序表达式的差异主要在于运算符所在位置不同,前序表达式是将运算符写在两个操作数之前,例如:

$$+ X Y$$

$$+ X * Y Z$$

后序表达式当然就是将运算符写在两个操作数之后,例如:

$$X Y +$$

$$X Y Z * +$$

将中序表达式转换成前序表达式或后序表达式的过程中,需要考虑到运算符的优先级,常见的运算符的优先级如下表:

优先级	运算符
<div style="text-align: center;"> 高 ↑ ↓ 低 </div>	括号:“(”,“)”
	负号“-”
	乘号“*”,除号“/”,余数“%”
	加号“+”,减号“-”
	等于类“<”,“<=”,“=”,“>=”,“>”

本小节先对中序转前序表达式做简单的说明,在后面小节中会再做更详细的介绍。

假设欲将中序表达式 $X / (Y - Z)$ 转换成前序表达式,其步骤如下:

步骤 1: 考虑表达式中最高优先权运算符,故先处理括号内的减法“-”,将其移至操作数“Y”、“Z”之前,如下所示:

$$X / (- Y Z)$$

步骤 2: 将括号视为独立的操作数来处理除法“/”,将其移至“X”、“(- Y Z)”之前,如下所示:

$$/ X (- Y Z)$$

步骤 3: 将表达式中的括号去掉即完成转换中序对前序的转换,如下所示:

$$/ X - Y Z$$

同样地,若欲将该中序表达式转换成后序表达式,其步骤如下:

$$X / (Y - Z)$$

步骤 1: 考虑表达式中最高优先权运算符，故先处理括号内的减法“-”，将其移至操作数“Y”、“Z”之后，如下所示：

$$X / (Y Z -)$$

步骤 2: 将括号视为独立的操作数来处理除法“/”，将其移至“X”、“(YZ-)”之前，如下所示：

$$X (Y Z -) /$$

步骤 3: 将表达式中的括号去掉即完成转换中序对前序的转换，如下所示：

$$X Y Z - /$$

由于前序表达式和后序表达式中不需要括号，因为其计算顺序只有一种，所以不需考虑操作数的优先级，故又称为无括号表达式表示法。

在说明中序表达式转成后序表达式前，先介绍如何使用堆栈的结果来计算前序、中序及后序表达式的结果。

4.5 中序表达式的表示法及计算

假设有一中序表达式如下：

$$X Y Z - /$$

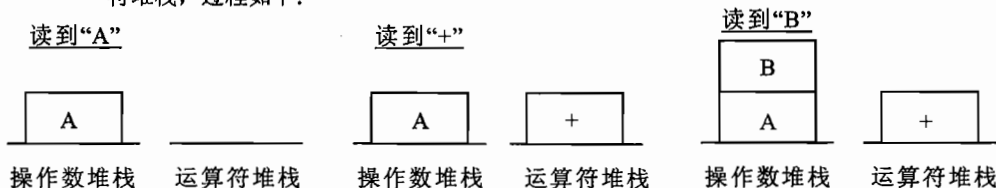
当我们对表达式从左至右读取时，因为要考虑运算符的优先级，故必须先计算“B/C”，而非“A+B”。在处理中序表达式的计算时，需要两个堆栈来存放运算符和操作数，如果一运算符的优先权高于下一个操作数时，则从操作数堆栈中取两个操作数与该运算符进行计算。现在我们将针对上述之中序表达式，以图来说明其计算的处理过程。

中序表达式：A + B / C - D (读取方向:由左至右)

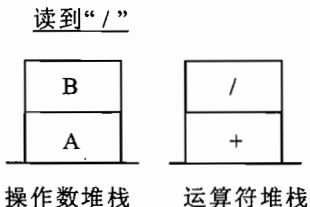
步骤 1: 建立操作数及运算符的堆栈，初始为空。

操作数堆栈 运算符堆栈

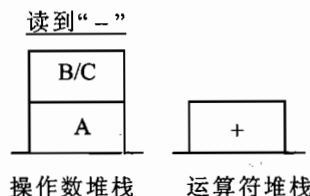
步骤 2: 从左至右读取表达式，如果读到操作数则置入操作数堆栈中；若读到运算符时则置入运算符堆栈，过程如下：



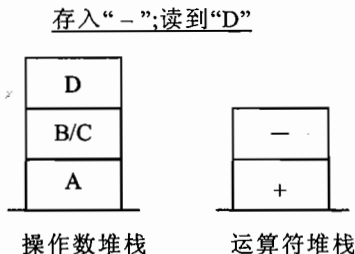
步骤 3: 当读到“/”时，由于“/”的优先权较运算符堆栈中的“+”高，则将“/”存入运算符堆栈。



步骤 4: 接着读到“-”时, 因为“-”的优先权较运算符堆栈中的“/”低, 故取出“/”及从操作数堆栈中取出两个操作数进行计算, 并将计算结果存回操作数堆栈。

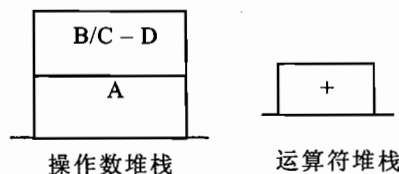


步骤 5: 然后再将刚刚未存入堆栈中的“-”存入运算符堆栈中, 最后读到“D”, 并将其存入操作数堆栈中。

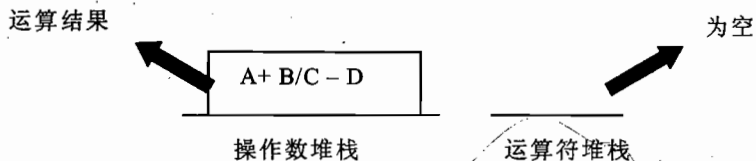


由于运算符堆栈中还有未计算的运算符, 故每取出一个运算符则同时从操作数堆栈中取出两个操作数做计算, 并将计算结果存回操作数堆栈中, 直到运算符堆栈为空时, 则完成表达式的计算, 且操作数堆栈中的内容即为表达式的计算结果。处理过程如下:

步骤 6: 取出运算符“-”及操作数“B/C”、“D”进行计算, 结果存回操作数堆栈如下:



步骤 7: 取出运算符“+”及操作数“B/C-D”、“A”进行计算, 结果存回操作数堆栈如下:



从上述的中序表达式的计算过程中, 可整理出其处理步骤如下:

1. 建立操作数和运算符的堆栈, 初始为空
2. 当表达式尚未读取完时: (由左至右)
 - 读取一个运算单元
 - (1) 若读取的是操作数, 则将其存入操作数堆栈
 - (2) 若读取的是运算符:
 - (a) 若运算符堆栈为空, 则将其存入运算符堆栈
 - (b) 若运算符堆栈非空, 则和运算符堆栈中的顶端运算符比较优先权, 若较堆栈中的高, 则直接存入堆栈中;
若较堆栈中的低:

- ① 从运算符堆栈中取出一个运算符
 - ② 从操作数堆栈中取出所需操作数
 - ③ 计算其值后将结果存回操作数堆栈
 - ④ 将刚刚读取的运算符存入运算符堆栈
3. 当运算符堆栈非空:
 - (1) 从运算符堆栈中取出一个运算符
 - (2) 从操作数堆栈中取出所需操作数
 - (3) 计算其值后将结果存回操作数堆栈
 4. 操作数堆栈的最后内容即为表达式的计算结果

程序实例:

输入一中序表达式, 并计算其值。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Stack_03.c */
03  /* 程序目的: 计算中序表达式 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07
08  struct s_node /*声明堆栈链接结构*/
09  {
10      int data; /*堆栈数据*/
11      struct s_node *next; /*链接指针*/
12  };
13  typedef struct s_node s_list; /*定义新类型列表*/
14  typedef s_list *link; /*定义新类型列表指针*/
15  link operator=NULL;
16  link operand=NULL;
17  /*-----*/
18  /*存入堆栈数据 */
19  /*-----*/
20  link push(link stack,int value)
21  {
22      link newnode; /*新节点指针*/
23
24      /*配置节点内存*/
25      newnode=(link) malloc (sizeof(s_list));
26      if (!newnode)
27      {
28          printf("\nMemory allocation failure!!!");
29          return NULL;
30      }
31      newnode->data=value; /*建立新节点*/
32      newnode->next=stack; /*将新节点指向原堆栈*/
33      stack=newnode; /*新节点为堆栈的顶端*/
34      return stack;
35  }
36  /*-----*/
37  /* 从堆栈中取出数据 */
38  /*-----*/
39  link pop(link stack,int *value)
40  {
41      link top; /*堆栈的顶端*/
42

```

```

43     if (stack !=NULL)
44     {
45         top=stack;           /*指向堆栈的顶端 */
46         stack=stack->next;    /*指向下一个节点 */
47         *value=top->data;     /*取出顶点数据 */
48         free(top);           /*释放节点空间 */
49         return stack;        /*返回堆栈指针 */
50     }
51     else
52         *value=-1;
53 }
54 /*-----*/
55 /*      检查堆栈是否为空      */
56 /*-----*/
57 int empty(link stack)
58 {
59     if (stack ==NULL)
60         return 1;           /*堆栈为空*/
61     else
62         return 0;           /*堆栈不为空*/
63 }
64 /*-----*/
65 /*      判断是否为运算符      */
66 /*-----*/
67 int is_operator(char operator)
68 {
69     switch (operator)
70     {
71         case '+': case '-': case '*': case '/': return 1; /*为运算符*/
72         default: return 0;
73     }
74 }
75 /*-----*/
76 /*      判断运算符的优先权      */
77 /*-----*/
78 int priority(char operator)
79 {
80     switch (operator)
81     {
82         case '+': case '-': return 1; /* "+"、 "-"运算符优先权值为 1*/
83         case '*': case '/': return 2; /* "*"、 "/"运算符优先权值为 2*/
84         default: return 0;
85     }
86 }
87 /*-----*/
88 /*      计算任两个操作数的值      */
89 /*-----*/
90 int two_result(int operator,int operand1,int operand2)
91 {
92     switch (operator)
93     {
94         case '+':return (operand2 + operand1);
95         case '-':return (operand2 - operand1);
96         case '*':return (operand2 * operand1);
97         case '/':return (operand2 / operand1);
98     }
99 }
100 /*-----*/
101 /*      主程序:输入中序表达式后计算出表达式的结果值      */
102 /*-----*/
103 void main( )

```

```

104 {
105     char expression[50];    /*声明表达式字符串数组*/
106     int position=0;         /*表达式位置*/
107     int op=0;               /*运算符*/
108     int operand1=0;         /*前操作数*/
109     int operand2=0;         /*后操作数*/
110     int evaluate=0;         /*运算结果*/
111
112     printf("\nPlease input the inorder expression :");
113     gets(expression);       /*读取中序表达式存入字符串数组 expression 中*/
114
115     while (expression[position]!='\0' && expression[position]!='\n')
116     {
117         if (is_operator(expression[position])) /*判断是否为运算符*/
118         {
119             if (!empty(operator))
120                 while (priority(expression[position]) <= priority(operator->data) && !
121                     empty(operator))
122                 {
123                     /*从堆栈中取出两个操作数和一个运算符*/
124                     operand=pop(operand,&operand1);
125                     operand=pop(operand,&operand2);
126                     operator=pop(operator,&op);
127                     operand=push(operand,two_result(op,operand1,operand2));
128                 }
129                 /*存入运算符堆栈*/
130                 operator=push(operator,expression[position]);
131             }
132             else
133                 /*存入操作数堆栈--需做 Ascii 码转换*/
134                 operand=push(operand,expression[position]-48);
135             position++;
136         }
137         /*取出运算符堆栈的内容*/
138         while (!empty(operator))
139         {
140             operator=pop(operator,&op);
141             operand=pop(operand,&operand1);
142             operand=pop(operand,&operand2);
143             /*计算后两操作数后存入堆栈*/
144             operand =push(operand,two_result(op,operand1,operand2));
145         }
146         /*取出表达式最终结果*/
147         operand=pop(operand,&evaluate);
148         printf("The expression [ %s] result is '%d'",expression,evaluate);
149     }

```

运行结果:

```

C:\DS>Stack_03
Please input the inorder expression : 8 * 9 - 4 * 2
The expression [8 * 9 - 4 * 2] result is '64'

C:\DS>

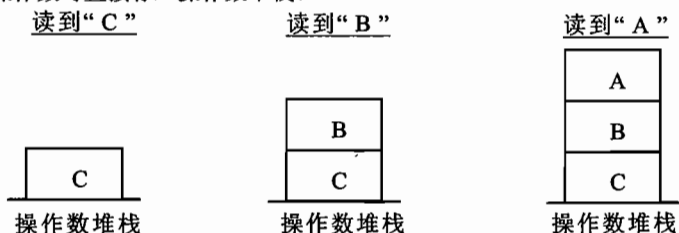
```


4.6 前序表达式的表示法及计算

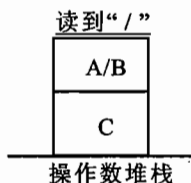
由于前序表达式是无括号的表达式表示法，也就是不用考虑优先权，故不需要运算符堆栈来暂存表达式中的运算符以用来比较优先级。计算前序表达式只需要一个操作数堆栈，特别要注意的是，读入前序表达式的方向和中序表达式相反，是要由右至左——读入，以下将说明前序表达式计算的处理过程。

前序表达式: $+ / A B C$ (读取方向: 由右至左)

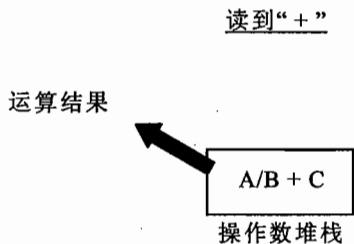
步骤 1: 读到操作数时直接存入操作数堆栈。



步骤 2: 读到“/”时则从操作数堆栈中取出两个操作数“A”、“B”做计算后将结果存回堆栈。



步骤 3: 读到“+”时则从操作数堆栈中取出两个操作数“A/B”、“C”做计算后将结果存回堆栈。



从上述的前序表达式的计算过程中，可整理出其处理步骤如下：

1. 建立操作数堆栈，初始为空
2. 当表达式尚未读取完时：(由右至左)
 - 读取一个运算单元
 - (1) 若读取的是操作数，则将其存入操作数堆栈
 - (2) 若读取的是运算符，则从操作数堆栈中取出所需操作数进行计算，并将计算结果存回堆栈
3. 当表达式读取完后，则操作数堆栈的内容即为表达式的计算结果

程序实例：

输入一前序表达式，并计算其值。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Stack_04.c */
03  /* 程序目的: 计算前序表达式 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07
08  struct s_node          /*声明堆栈链接结构*/
09  {
10      int data;          /*堆栈数据*/
11      struct s_node *next; /*链接指针*/
12  };
13  typedef struct s_node s_list; /*定义新类型列表*/
14  typedef s_list *link; /*定义新类型列表指针*/
15  link prefix=NULL;
16  link operand=NULL;
17  /*-----*/
18  /*      存入堆栈数据      */
19  /*-----*/
20  link push(link stack,int value)
21  {
22      link newnode; /*新节点指针*/
23
24      /*配置节点内存*/
25      newnode=(link) malloc (sizeof(s_list));
26      if (!newnode)
27      {
28          printf("\nMemory allocation failure!!!");
29          return NULL;
30      }
31      newnode->data=value; /*建立新节点*/
32      newnode->next=stack; /*将新节点指向原堆栈*/
33      stack=newnode; /*新节点为堆栈之顶端*/
34      return stack;
35  }
36  /*-----*/
37  /*      从堆栈中取出数据      */
38  /*-----*/
39  link pop(link stack,int *value)
40  {
41      link top; /*堆栈的顶端*/
42
43      if (stack !=NULL)
44      {
45          top=stack; /*指向堆栈的顶端*/
46          stack=stack->next; /*指向下一个节点*/
47          *value=top->data; /*取出顶点数据*/
48          free(top); /*释放节点空间*/
49          return stack; /*返回堆栈指针*/
50      }
51      else
52          *value=-1;
53  }
54  /*-----*/
55  /*      检查堆栈是否为空      */
56  /*-----*/
57  int empty(link stack)
58  {

```

```

59     if (stack ==NULL)
60         return 1;    /*堆栈为空*/
61     else
62         return 0;    /*堆栈不为空*/
63 }
64 /*-----*/
65 /*      判断是否为运算符      */
66 /*-----*/
67 int is_operator(char operator)
68 {
69     switch (operator)
70     {
71         case '+': case '-': case '*': case '/': return 1; /*为运算符*/
72         default: return 0;
73     }
74 }
75 /*-----*/
76 /*      计算任两个操作数的值      */
77 /*-----*/
78 int two_result(int operator,int operand1,int operand2)
79 {
80     switch (operator)
81     {
82         case '+':return (operand2 + operand1);
83         case '-':return (operand2 - operand1);
84         case '*':return (operand2 * operand1);
85         case '/':return (operand2 / operand1);
86     }
87 }
88 /*-----*/
89 /*主程序:输入前序表达式后计算出表达式之结果值      */
90 /*-----*/
91 void main( )
92 {
93     char expression[50];    /*声明表达式字符串数组*/
94     int position=0;        /*表达式位置*/
95     int operand1=0;        /*前操作数*/
96     int operand2=0;        /*后操作数*/
97     int evaluate=0;        /*运算结果*/
98     int token=0;          /*运算符或操作数*/
99
100     printf("\nPlease input the preorder expression :");
101     gets(expression);      /*读取前序表达式存入字符串数组 expression 中*/
102
103     while (expression[position]!='\0' && expression[position]!='\n')
104     {
105         prefix=push(prefix,expression[position]);
106         position++;
107     }
108
109     /*取出运算符堆栈的内容*/
110     while (!empty(prefix))
111     {
112         prefix=pop(prefix,&token); /*先取出一个数据值*/
113         if (is_operator(token))    /*判断是否为运算符*/
114         {
115             operand=pop(operand,&operand1);
116             operand=pop(operand,&operand2);
117             /*计算后两操作数后存入堆栈*/
118             operand =push(operand,two_result(token,operand2,operand1));
119         }

```

```

120         else
121             /*存入操作数堆栈--需做Ascii码转换*/
122             operand=push(operand,token-48);
123     }
124     /*取出表达式最终结果*/
125     operand=pop(operand,&evaluate);
126     printf("The expression [ %s] result is '%d'",expression,evaluate);
127 }

```

运行结果:

```

C:\DS>Stack_04
Please input the inorder expression : - * 6 4 * 2 3
The expression [ - * 6 4 * 2 3 ] result is '18'
C:\DS>

```

4.7 后序表达式的表示法及计算

后序表达式的计算处理过程和前序表达式相同,唯一不同的是读取表达式的方向和前序表达式相反,而是由左至右读取,详细的过程可参考上一节的前序表达式的方法,其步骤如下:

1. 建立操作数堆栈,初始为空
2. 当表达式尚未读取完时:(由左至右)
 - 读取一个运算单元
 - (1) 若读取的是操作数,则将其存入操作数堆栈
 - (2) 若读取的是运算符,则从操作数堆栈中取出所需操作数进行计算,并将计算结果存回堆栈
3. 当表达式读取完后,则操作数堆栈的内容即为表达式的计算结果

程序实例:

输入一后序表达式,并计算其值。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Stack_05.c */
03  /* 程序目的: 计算后序表达式 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07
08  struct s_node /*声明堆栈链接结构*/
09  {
10      int data; /*堆栈数据*/
11      struct s_node *next; /*链接指针*/
12  };
13  typedef struct s_node s_list; /*定义新类型列表*/
14  typedef s_list *link; /*定义新类型列表指针*/
15
16  link operand=NULL;
17  /*-----*/
18  /* 存入堆栈数据 */
19  /*-----*/
20  link push(link stack,int value)
21  {
22      link newnode; /*新节点指针*/

```

```

23
24             /*配置节点内存*/
25     newnode=(link) malloc (sizeof(s_list));
26     if (!newnode)
27     {
28         printf("\nMemory allocation failure!!!");
29         return NULL;
30     }
31     newnode->data=value;    /*建立新节点*/
32     newnode->next=stack;    /*将新节点指向原堆栈*/
33     stack=newnode;         /*新节点为堆栈的顶端*/
34     return stack;
35 }
36 /*-----*/
37 /*      从堆栈中取出数据      */
38 /*-----*/
39 link pop(link stack,int *value)
40 {
41     link top;    /*堆栈的顶端*/
42
43     if (stack !=NULL)
44     {
45         top=stack;    /*指向堆栈的顶端*/
46         stack=stack->next;    /*指向下一个节点*/
47         *value=top->data;    /*取出顶点数据*/
48         free(top);    /*释放节点空间*/
49         return stack;    /*返回堆栈指针*/
50     }
51     else
52         *value=-1;
53 }
54 /*-----*/
55 /*      检查堆栈是否为空      */
56 /*-----*/
57 int empty(link stack)
58 {
59     if (stack ==NULL)
60         return 1;    /*堆栈为空*/
61     else
62         return 0;    /*堆栈不为空*/
63 }
64 /*-----*/
65 /*      判断是否为运算符      */
66 /*-----*/
67 int is_operator(char operator)
68 {
69     switch (operator)
70     {
71         case '+': case '-': case '*': case '/': return 1; /*为运算符*/
72         default: return 0;
73     }
74 }
75 /*-----*/
76 /*      计算任两个操作数的值      */
77 /*-----*/
78 int two_result(int operator,int operand1,int operand2)
79 {
80     switch (operator)
81     {
82         case '+':return (operand2 + operand1);
83         case '-':return (operand2 - operand1);

```

```

84     case '*':return (operand2 * operand1);
85     case '/':return (operand2 / operand1);
86 }
87 }
88 /*-----*/
89 /*主程序:输入后序表达式后计算出表达式之结果值 */
90 /*-----*/
91 void main( )
92 {
93     char expression[50]; /*声明表达式字符串数组*/
94     int position=0;      /*表达式位置*/
95     int operand1=0;      /*前操作数*/
96     int operand2=0;      /*后操作数*/
97     int evaluate;
98
99     printf("\nPlease input the postorder expression :");
100    gets(expression);    /*读取后序表达式存入字符串数组 expression 中*/
101
102    while (expression[position]!='\0' && expression[position]!='\n')
103    {
104        if (is_operator(expression[position])) /*判断是否为运算符*/
105        {
106            operand=pop(operand,&operand1);
107            operand=pop(operand,&operand2);
108            /*计算后两操作数后存入堆栈*/
109            operand=push(operand,two_result(expression[position],
110            operand1,operand2));
111        }
112        else
113            /*存入操作数堆栈--需做 Ascii 码转换*/
114            operand=push(operand,expression[position]-48);
115        position++;
116    }
117    /*取出表达式最终结果*/
118    operand=pop(operand,&evaluate);
119    printf("The expression [ %s] result is '%d'",expression,evaluate);
120 }

```

运行结果:

```

C:\DS>Stack_05
Please input the postorder expression : 9 2 * 4 2 / -
The expression [ 9 2 * 4 2 / - ] result is '16'

C:\DS>

```

4.8 表达式的转换

介绍完前序、中序及后序表达式后可以了解使用前序及后序表达式由于不需考虑运算符的优先级，因此计算方法较容易。但是我们一般常用的表达式为中序表达式，为方便计算机处理，我们希望能将中序表达式转换成后序表达式。

假设有一个中序表达式，将其转换成后序表达式的原则如下：

1. 从左至右读取一中序表达式。
2. 若读取的是操作数，则直接打印。
3. 若读取的是运算符：
 - (1) 该运算符为左括号“(”，则直接存入堆栈。
 - (2) 该运算符为右括号“)”，则输出堆栈中的右运算符，直到取出左括号为止。
 - (3) 该运算符为非括号运算符，则与堆栈顶端的运算符做优先权比较：
 - (a) 若较堆栈顶端运算符高或相等，则直接存入堆栈。
 - (b) 若较堆栈顶端运算符低，则输出堆栈中的运算符。
4. 当表达式读取完成后堆栈中尚有运算符时，则依序取出运算符，直到堆栈为空。

例如欲将下列的中序表达式转换成后序表达式：

中序表达式： $A / B - C * (D + E)$ (读取方向：由左至右)

步骤 1：读到“A”，直接输出

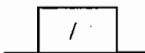
[后序表达式：]

A

步骤 2：读到“/”，存入堆栈

[后序表达式：]

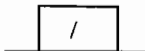
A



步骤 3：读到“B”，直接输出

[后序表达式：]

A B



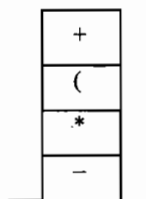
步骤 4：读到“-”和堆栈的顶端比较，由于“-”的优先权较低，故先取出堆栈顶端运算符后再存入堆栈。

[后序表达式:]
A B /



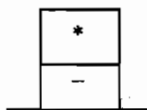
步骤 5: 依序处理, 直到读到“E”时结果如下:

[后序表达式:]
A B / C D E



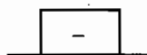
步骤 6: 当读到右括号“)”时, 则将堆栈中的运算符输出, 直到取出左括号“(”。

[后序表达式:]
A B / C D E +

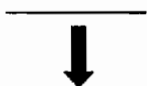


步骤 7: 此时已读取完中序表达式, 但堆栈中仍有运算符, 则依序输出运算符, 直到堆栈为空为止。

[后序表达式:]
A B / C D E + *



[后序表达式:]
A B / C D E + * -



堆栈为空



完成表达式转换

下面的范例是将中序表达式转换成后序表达式的完整程序, 至于中序表达式转换成前序表达式的处理和后序表达式相似, 就留给读者自行练习。

程序实例:

中序表达式转后序表达式

程序构思:

从左至右读取一个中序表达式。若读取的是操作数, 则直接输出。若读取的是运算符: 该运算符为左括号“(”, 则直接存入堆栈。如果该运算符为右括号“)”, 则输出堆栈中的运算符, 直到取出左括号为止。如果该运算符为非括号运算符, 则与堆栈顶端的运算符做优先权比较: 若较堆栈顶端运算符高或相等, 则直接存入堆栈。若较堆栈顶端运算符低, 则输出堆栈中的运算符。如果

表达式读取完成，而堆栈中尚有运算符时，则依序取出运算符，直到堆栈为空。

程序源代码：

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Stack_06.c */
03  /* 程序目的: 中序表达式转后序表达式 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  struct s_node /*声明堆栈链接结构*/
08  {
09      int data; /*堆栈数据*/
10      struct s_node *next; /*链接指针*/
11  };
12  typedef struct s_node s_list; /*定义新类型列表*/
13  typedef s_list *link; /*定义新类型列表指针*/
14  link operator=NULL; /*运算符堆栈*/
15  /*-----*/
16  /* 检查堆栈是否为空 */
17  /*-----*/
18  int empty(link stack)
19  {
20      if (stack == NULL)
21          return 1; /*堆栈为空 */
22      else
23          return 0; /*堆栈不为空*/
24  }
25  /*-----*/
26  /* 判断是否为运算符 */
27  /*-----*/
28  int is_op(char op)
29  {
30      switch (op)
31      {
32          case '(':
33          case ')':
34          case '+':
35          case '-':
36          case '*':
37          case '/': return 1;
38          default: return 0;
39      }
40  }
41  /*-----*/
42  /* 判断运算符的优先权 */
43  /*-----*/
44  int priority(char op)
45  {
46      switch (op)
47      {
48          case '(': return 1;
49          case '+':
50          case '-': return 2;
51          case '*':
52          case '/': return 3;
53          default: return 0;
54      }
55  }
56

```

```

57  /*-----*/
58  /*      存入堆栈数据      */
59  /*-----*/
60  link push(link stack,int value)
61  {
62      link newnode;    /*声明新节点指针*/
63
64      /*配置节点内存*/
65      newnode=(link) malloc (sizeof(s_list));
66      newnode->data=value;    /*建立新节点*/
67      newnode->next=stack;    /*将新节点指向原堆栈*/
68      stack=newnode;        /*新节点为堆栈之顶端*/
69      return stack;
70  }
71
72  /*-----*/
73  /*      从堆栈中取出数据      */
74  /*-----*/
75  link pop( link stack,int *value)
76  {
77      link top;        /*堆栈的顶端*/
78
79      if (stack !=NULL)
80      {
81          top=stack;    /*指向堆栈的顶端*/
82          stack=stack->next; /*指向下一个节点*/
83          *value=top->data; /*取出顶点数据*/
84          free(top);    /*释放节点空间*/
85          return stack; /*返回堆栈指针*/
86      }
87      else
88          *value= -1;
89  }
90
91  /*-----*/
92  /*主程序:输入中序表达式,经转换后输出后序表达式      */
93  /*-----*/
94  void main ( )
95  {
96      char inorder[50];    /*声明中序表达式字符串数组*/
97      char postorder[50]; /*转换的后序表达式*/
98      int op=0;            /*运算符*/
99      int in_position;     /*中序表达式所在位置*/
100     int po_position;     /*后序表达式所在位置*/
101     int i;
102
103     in_position=0;
104     po_position=0;
105     for (i=0;i<=50;i++) postorder[i]=' '; /*清除数组内容*/
106     printf("Please input the inorder expression:");
107     gets(inorder);    /*读取中序表达式*/
108
109     while (inorder[in_position]!='\0' && inorder[in_position]!='\n')
110     {
111         if (is_op(inorder[in_position])) /*判断是否为运算符*/
112         {
113             if (empty(operator) || inorder[in_position]=='(')
114                 /*将运算符存到堆栈中*/
115                 operator=push(operator,inorder[in_position]);
116             else
117                 if (inorder[in_position] == ')')

```

```

118         {
119             while (operator->data != '(' )    /*取出运算符直到取出' (' */
120             {
121                 operator=pop(operator,&op);
122                 /*存到后序表达式数组*/
123                 postorder[po_position++]=op;
124             }
125         }
126         else
127         {
128             while (priority(inorder[in_position]) <= priority(operator->data) && !
129                 empty(operator))
130             {
131                 operator=pop(operator,&op);
132                 postorder[po_position++]=op; /*存到后序表达式数组*/
133             }
134             operator=push(operator,inorder[in_position]);
135         }
136     }
137 }
138 else
139     postorder[po_position++]=inorder[in_position];
140     in_position++;
141 }
142 while (!empty(operator))    /*取出在堆栈中所有的运算符*/
143 {
144     operator=pop(operator,&op);
145     postorder[po_position++]=op;    /*存到后序表达式数组*/
146 }
147 postorder[po_position-1]='\0';
148 printf("The postorder expression is [");
149 for (i=0;i<50;i++)
150     if (postorder[i]!='(' && postorder[i]!='\0' )
151         printf("%c ",postorder[i]);
152     printf("\n");
153 }

```

运行结果:

```

C:\DS>Stack_06
Please input the inorder expression: 8 / 2 + ( 7 - 2 ) * 4

The postorder expression is [ 8 2 / 7 2 - 4 * + ]

C:\DS>

```

【习题】

一、复习:

1. 请说明何谓“堆栈”? 并举出 3 个堆栈的应用。
2. 堆栈的建立可使用两种结构: _____ 结构和 _____ 结构。
3. 堆栈的数据存取特性为_____。
4. 若将 5, 4, 3, 2, 1 依序存入堆栈中, 则将数据从堆栈取出至堆栈为空的顺序为_____。

二、应用:

1. 试比较数组堆栈和链表堆栈的差异。
2. 试计算下列表达式的值:
 - (1) 前序表达式: $**76-/821$
 - (2) 中序表达式: $8*9-(16-4)/2+5$
 - (3) 后序表达式: $576*82+5/-*$
3. 将下列中序表达式转换成后序表达式,并绘出堆栈的处理过程.
 - (1) $A/B*C-A/C*F$
 - (2) $(A+B*C)+(B/C-E)$
4. 进行下列表达式之转换
 - (1) 前序: $/+-+AB*C+DEF+G*HI$, 转中序、后序
 - (2) 中序: $(A+B)*D+E/(F+A*D)+C$, 转前序、后序
 - (3) 后序: $AC-D/EAB*F+/+$
5. 试说明为何要将中序表达式转换成前序表达式或后序表达式。
6. 请写出中序表达式转换成后序表达式的步骤。
7. 试写出一个程序将中序表达式转换成前序表达式。
8. 试写出一个算法将后序表达式转换成前序表达式。

队 列

第 5 章

- ◆ 何谓队列
- ◆ 用数组仿真队列
- ◆ 用链表仿真队列
- ◆ 环状队列
- ◆ 双向队列

在第 4 章中提到“堆栈”和“队列”(Queue)是在程序设计时经常使用的数据结构,两者都是有序列表,只是对于其内含数据之存取方式不同。介绍完“堆栈”的建立及应用后,本章将对“队列”做详细的说明。

5.1 何谓队列

队列数据结构规定:在有序列表中数据的输出、输入是分别由不同端进行处理,输出端称为前端(front),输入端称为后端(rear),这样会使得先存入的数据会先被取出,也就是具有先进先出 FIFO(First In First Out)的特性。

队列的应用也很多,下面列出几项较常见的:

1. 图形的广度优先(Breadth-first)搜索法。
2. 优先队列(Priority Queue),此种队列在取出元素时是根据所存元素的某项特性值或优先权而取出具最小或最大数值的元素。
3. 操作系统中的工作调度,若工作的优先权相同,则采用先到先做的原则。
4. 用于“spooling”,先将输出数据写在磁盘上,再由打印机把先存入者先处理的顺利将数据输出。

队列和堆栈一样也可使用两种结构:数组结构和链表结构,将分别在 5.2 节和 5.3 节做介绍。

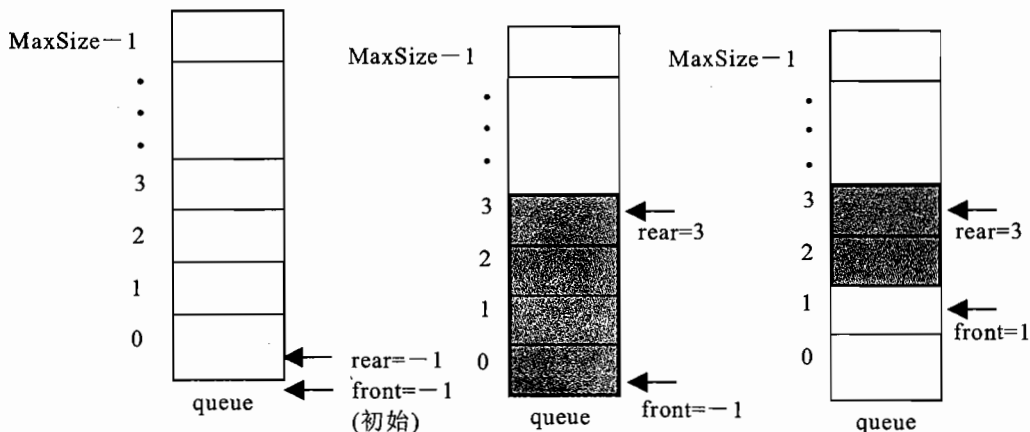
5.2 用数组仿真队列

队列本身是有序列表,若使用数组的结构来存储队列的数据,则队列数组的声明如下:

```
int queue[MaxSize];
int front=-1;
int rear=-1;
```

其中 MaxSize 是该队列的最大容量。因为队列的输出、输入是分别从前后端来处理,因此需要两个变量 front 及 rear 分别记录队列前后端的索引值,front 会随着数据输出而变动,而 rear 则是随着数据输入而改变。

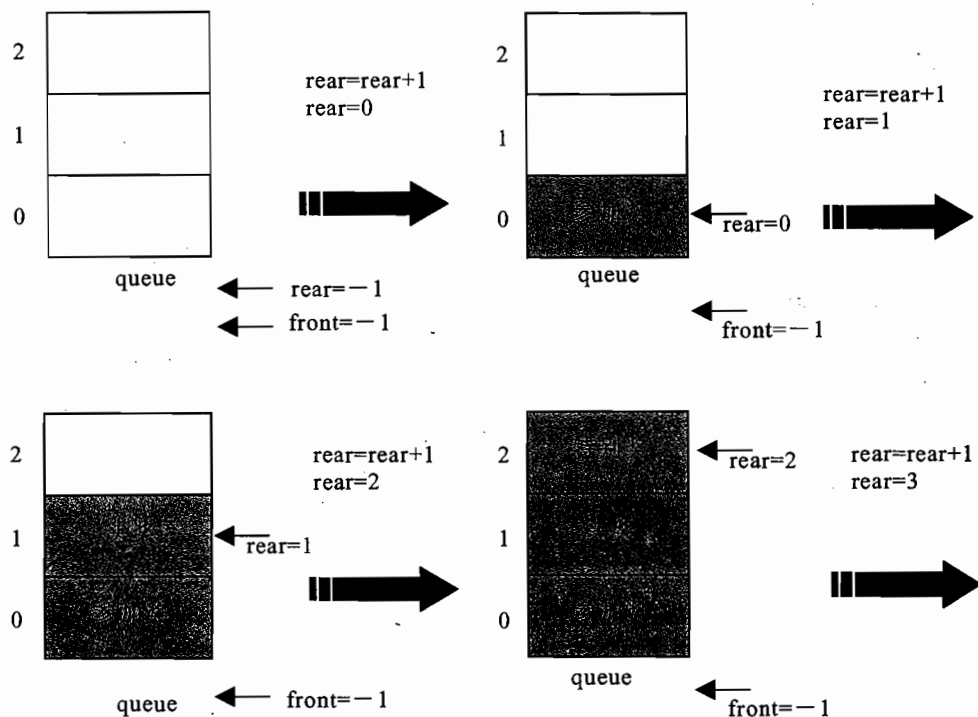
队列的数组结构如下图所示:



当我们将数据存入队列时称为“addqueue”，addqueue 的处理主要有两个步骤：

- (1) 将队尾指针往前移： $\text{rear} + 1$
- (2) 若尾指针 rear 小于等于队列的最大索引值 $\text{MaxSize} - 1$ ，则将数据存入 rear 所指的数组元素中，否则无法存入数据。

例如百货公司举行抽奖活动，欲以队列的方式将 4 个奖项置入抽奖台，依序为“参加奖”、“头奖”、“特别奖”及“神秘奖”，则 addqueue 的处理如下：

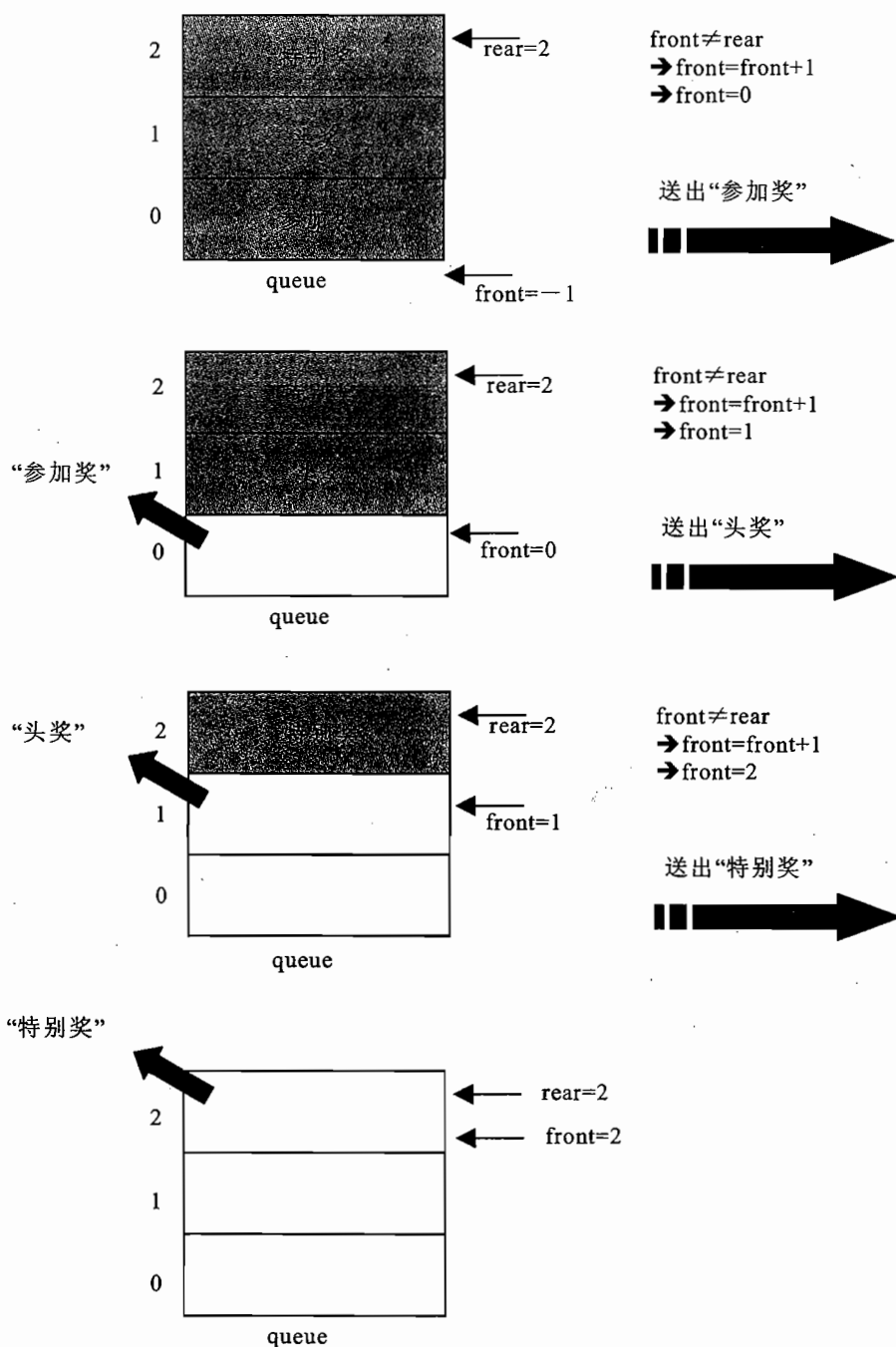


此时尾指针 rear 已经大于 $\text{MaxSize} - 1$ ，故无法继续置入“神秘奖”。

另外，从队列中取出数据项称为“delqueue”，delqueue 的操作可分为 4 个步骤：

- (1) 检查队列中是否有数据存在。
- (2) 若头指针 front 等于尾指针 rear ，则表示队列中无数据。
- (3) 若头指针 front 不等于尾指针 rear ，则将队头指针往前移 $\text{front} + 1$ 。
- (4) 取出队头指针所指的数组元素内容。

例如欲将抽奖台的 3 个奖项依序送出，delqueue 的处理如下：



此时队头指针 front 等于尾指针 rear，则表示队列已为空，故停止输出。

从最后一个队列中可看到，当尾指针 rear 等于 MaxSize-1 时，虽然队列中仍有空位却已经无法存入数据，如此一来则不能有效地利用空间。对于这种状况可使用“环状队列”的方式来解决，我们将在 5.4 节再做介绍。

程序实例:

队列数据的存取(使用数组)。

程序构思:

数据输入时先将队尾指针往前移 $rear + 1$, 若 $rear$ 小于或等于队列的最大索引值 $MaxSize - 1$, 则将数据存入 $rear$ 所指的数组元素中, 否则无法存入数据。而输出数据前先检查队列中是否有数据存在, 若头指针 $front$ 等于尾指针 $rear$, 则表示队列中无数据, 若头指针 $front$ 不等于尾指针 $rear$, 则将队头指针往前移 $front + 1$, 取出队头指针所指的数组元素内容。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Queue_01.c */
03  /* 程序目的: 队列数据之存取(使用数组) */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  # define MaxSize 10          /*队列的最大容量 */
08  int queue[ MaxSize];        /*声明队列数组 */
09  int front = -1;              /*初始队头指针 */
10  int rear = -1;              /*初始队尾指针 */
11  /*-----*/
12  /*          输入队列数据          */
13  /*-----*/
14  void addqueue (int value)
15  {
16      if (rear >= MaxSize)      /*检查队列是否已满 */
17          printf("The queue is full!!"); /*无法存入数据 */
18      else
19      {
20          rear++;              /*尾指针往前进 */
21          queue[rear] = value; /*将数据存入队列*/
22      }
23  }
24  /*-----*/
25  /*          输出队列数据          */
26  /*-----*/
27  int delqueue ( )
28  {
29      int temp;
30
31      if ( front == rear)      /*检查队列是否已空*/
32          return -1;
33      else
34      { front++;              /*头指针往前进*/
35        temp=queue[front];
36        queue[front]=0;
37        return temp;          /*输出队列数据*/
38      }
39  }
40  /*-----*/
41  /* 进行队列数据之输出输入, 并打输出队列内容 */
42  /*-----*/
43  void main ( )
44  {

```

```

45     int select ;                /*功能选项变量 */
46     int i,temp ;
47
48     for (i=0;i<=MaxSize;i++)    /*清除数组内容*/
49         queue[i]=0;
50
51     printf("(1)Input a data\n");
52     printf("(2)Output a data\n");
53     printf("(3)Exit\n");
54     printf("Please select:");
55     scanf ( "%d", &select) ;
56     printf("\n");
57     if (select!=3)
58     {
59         do
60         {
61             switch ( select )
62             {
63                 case 1 : printf ( "Please input the input value == > " ) ;
64                           scanf("%d" ,&temp) ;                /*读取输入值*/
65                           addqueue(temp);                    /*将数据插入队列中*/
66                           printf("The content of queue:"); /*输出队列内容*/
67                           for (i=0;i<=MaxSize;i++)
68                           {
69                               if (queue[i]!=0)
70                                   printf("[%d]",queue[i]);
71                           }
72                           break;
73                 case 2 : if ( (temp = delqueue () ) == -1 ) /*判断队列是
74                           否为空*/
75                           printf ("The queue is empty!!! \n ");
76                           else
77                           {
78                               printf ( "The output value is [%d]\n" , temp );
79                               printf("The content of queue:"); /*输出队列内容*/
80                               for (i=0;i<=MaxSize;i++)
81                               {
82                                   if (queue[i]!=0)
83                                       printf("[%d]",queue[i]);
84                               }
85                           }
86                           break ;
87             }
88             printf("\n(1)Input a data\n");
89             printf("(2)Output a data\n");
90             printf("(3)Exit\n");
91             printf("Please select:");
92             scanf ( "%d", &select) ;
93             printf("\n");
94         }while (select !=3);
95     }
96

```

运行结果:

```

C:\DS>Queue_01
(1)Input a data
(2)Output a data
(3)Exit
Please select:1

```

```
Please input the input value=> 1
```

```
The content of queue: [1]
```

```
(1)Input a data  
(2)Output a data  
(3)Exit
```

```
Please select:2
```

```
The output value is [1]
```

```
The content of queue:
```

```
(1)Input a data  
(2)Output a data  
(3)Exit
```

```
Please select:2
```

```
The queue is empty!!
```

```
(1)Input a data  
(2)Output a data  
(3)Exit
```

```
Please select:3
```

```
C:\DS>
```

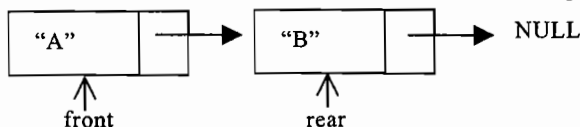
5.3 用链表仿真队列

除了使用数组结构外,也可以使用链表结构来建立队列。队列的链表结构如下:

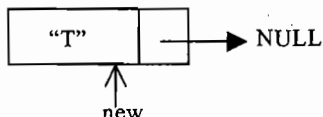
```
struct q_node  
{ int data;  
  struct q_node *point;  
}  
typedef struct q_node s_list;  
typedef q_list *linklist;  
linklist front=NULL;  
linklist rear=NULL;
```

其中 front 和 rear 是作为队列前后端的输出、输入指针的控制,由于刚刚建立的队列为空,故先将 front 和 rear 指向 NULL。

假设已知一个队列链表如下,欲插入一新数据项“T”到队列的函数 addqueue() 的处理,其步骤为:

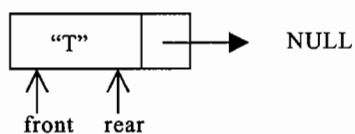


步骤 1: 建立一个新节点后存入新数据项内容“T”



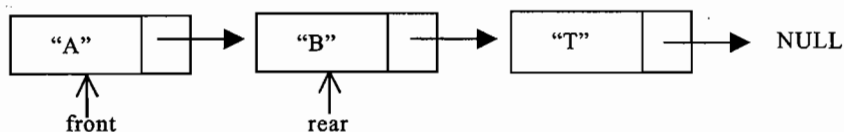
步骤 2: 判断队尾指针 rear 是否为 NULL

(1) 若尾指针 rear 为 NULL,则此新节点为队列的第 1 个数据,将 front 及 rear 都指向新节点。

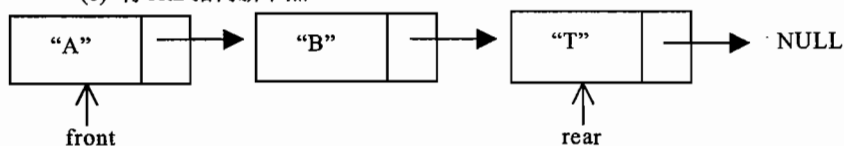


(2) 若尾指针 rear 不为 NULL

(a) 将 rear 所指节点的指针指向新节点

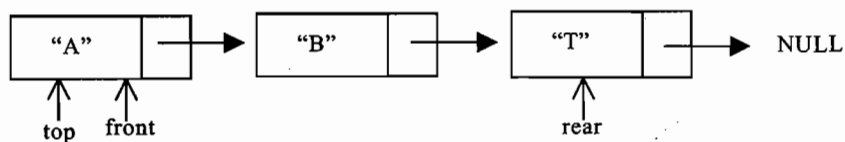


(b) 将 rear 指向新节点

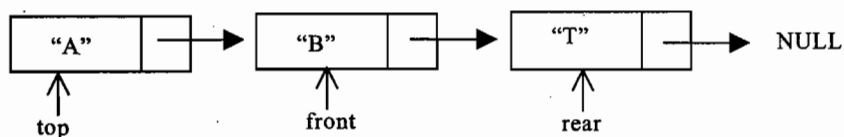


队列数据的输入是从后端 rear 进行，而数据输出则是从队列的前端 front 处理，其步骤如下：

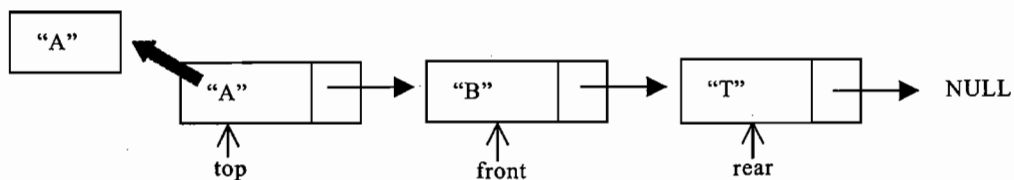
步骤 1：先保留队头指针 front 所指的位置



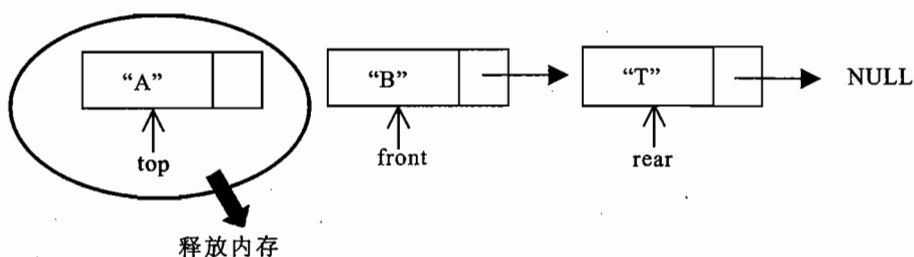
步骤 2：将头指针 front 指向下一个节点



步骤 3：取出之前保留头指针所指的节点内容
data



步骤 4：释放原队列前端所指的节点内存



程序实例:

队列数据之存取(使用链表)。

程序构思:

输入数据时要先判断 rear 是否为 NULL, 若 rear 为 NULL, 则此新节点为队列的第一个数据, 将 front 及 rear 均指向新节点。若 rear 不为 NULL, 先将 rear 所指节点的指针指向新节点, 再将 rear 指向新节点。而数据输出要先保留队头指针 front 所指的位置, 将头指针 front 指向下一个节点后, 取出当前保留头指针所指的节点内容, 最后再释放原队列前端所指的节点内存。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Queue_02.c */
03  /* 程序目的: 队列数据的存取(使用链表) */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  struct queue_node /*声明队列链接结构*/
08  {
09      int data;
10      struct queue_node *next;
11  };
12  typedef struct queue_node queue_list; /*定义队列类型 */
13  typedef queue_list *link; /*定义队列指针类型 */
14  link front = NULL; /*初始队头指针 */
15  link rear = NULL; /*初始队尾指针 */
16  /*-----*/
17  /*      输入队列数据      */
18  /*-----*/
19  void addqueue (int value)
20  {
21      link new_node;
22
23      new_node = ( link ) malloc (sizeof(queue_list) );
24      new_node->data = value; /*将数据存入队列*/
25      new_node->next= NULL; /*设置初值*/
26      if (rear==NULL) /*若为队列的第一个数据*/
27          front=new_node; /*将 front 指向新节点*/
28      else
29          rear->next =new_node; /*rear 所指的节点指向新节点*/
30          rear=new_node; /*rear 指向新节点*/
31  }
32  /*-----*/
33  /*      输出队列数据      */
34  /*-----*/
35  int delqueue ()
36  {
37      link top;
38      int temp;
39      if ( front !=NULL) /*检查队列是否已空*/
40      {
41          top = front ; /*将 top 指向 front*/
42          front = front -> next; /*删除之前节点 */
43          temp = top -> data ; /*暂存输出队列数据*/
44          free (top); /*释放输出节点的内存*/
45          return temp; /*输出队列数据*/

```

```

46     }
47     else
48         return -1 ;
49     }
50     /*-----*/
51     /* 进行队列数据之输出输入, 并打输出队列内容 */
52     /*-----*/
53     void main ( )
54     {
55         int select ;      /*功能选项变量 */
56         int i,temp ;
57         link point;
58
59         printf("(1)Input a data\n");
60         printf("(2)Output a data\n");
61         printf("(3)Exit\n");
62         printf("Please select:");
63         scanf ( "%d", &select) ;
64         printf("\n");
65         if (select!=3)
66         {
67             do
68             {
69                 switch ( select )
70                 {
71                     case 1 :printf ( "Please input the input value == > " ) ;
72                             scanf("%d",&temp) ;      /*读取输入值*/
73                             addqueue(temp);          /*将数据插入队列中*/
74                             printf("The content of queue:");    /*输出队列内容*/
75                             point=front;
76                             while (point!=NULL)
77                             {
78                                 printf("[%d]",point->data);
79                                 point=point->next;
80                             }
81                             break;
82                     case 2 : if ((temp = delqueue()) == -1 ) /*判断队列是否为空*/
83                             printf ("The queue is empty!!! \n ");
84                             else
85                             {
86                                 printf ( "The output value is [%d]\n" , temp );
87                                 printf("The content of queue:");    /*输出队列内容*/
88                                 point=front;
89                                 while (point!=NULL)
90                                 {
91                                     printf("[%d]",point->data);
92                                     point=point->next;
93                                 }
94                             }
95                             break ;
96                 }
97                 printf("\n(1)Input a data\n");
98                 printf("(2)Output a data\n");
99                 printf("(3)Exit\n");
100                printf("Please select:");
101                scanf ( "%d", &select) ;
102                printf("\n");
103            }while (select !=3);
104        }
105    }

```

运行结果:

```
C:\DS>Queue_02
(1)Input a data
(2)Output a data
(3)Exit
Please select:1
Please input the input value=> 1
The content of queue: [1]

(1)Input a data
(2)Output a data
(3)Exit
Please select:2
The output value is [1]
The content of queue:

(1)Input a data
(2)Output a data
(3)Exit
Please select:2

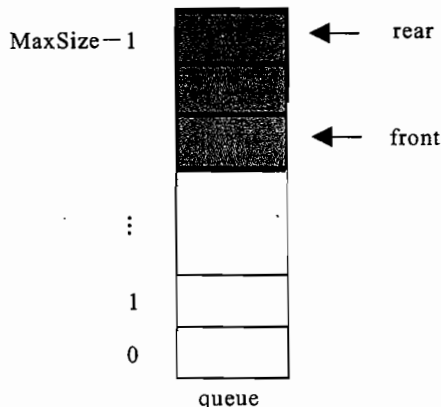
The queue is empty!!

(1)Input a data
(2)Output a data
(3)Exit
Please select:3

C:\DS>
```

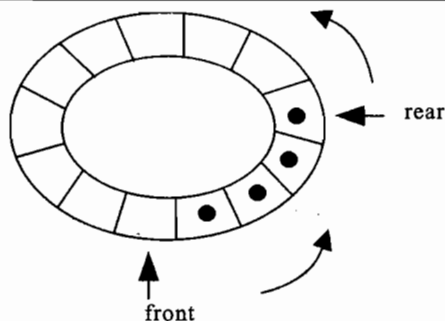
5.4 环状队列

我们在 5.2 节中提到，当队尾指针 $rear$ 等于 $MaxSize$ 时，不论队列是否有空间，都无法再将数据存于队列中，如下图：



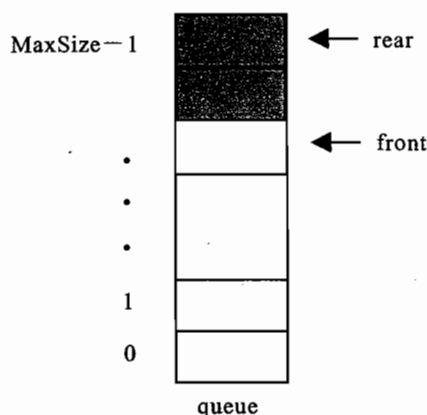
如果要将数据往队列前端移动以挪出空间，需要花费很多时间。为解决这样的问题，我们使用环状队列，控制队列前尾指针 $front$ 、 $rear$ 来充分运用队列中的空间。

环状队列的概念如下图：



当插入数据时，尾指针 $rear$ 会往箭头方向前进，而输出数据时，头指针 $front$ 也会往箭头方向前进，可看出队列空间的利用是按照逆时针方向循环，直到 $rear$ 往前移动到等于 $front$ 时，表示队列已满始无法输入数据。

从上图看来，虽然环状队列看似一个封闭的圆环，但事实上环状队列仍然是一个线性数组，和一般队列比较起来，主要是前后端使用技巧的差异。在此需特别注意的是，环状队列中的头指针所指的数组位置并没有内容值存在，输出的值为 $front$ 的下一个元素，故环状队列实际上所能使用的空间为 $MaxSize-1$ 。



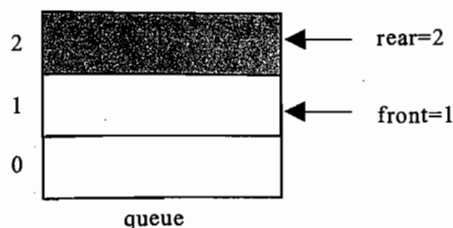
当尾指针 $rear$ 等于 $MaxSize-1$ 时需回到队列的最前端，故当输入数据时， $rear$ 所指的数组元素索引值采用下列的计算方法：

$$(rear + 1) \bmod MaxSize$$

若尾指针 $rear$ 不断前进直到等于头指针 $front$ 时，那么表示队列已满，但如果队列为空时 $rear$ 也等于 $front$ ，为区分这两种状况，必须使用不同的条件来加以判断。

$$\left[\begin{array}{l} \text{当队列为满: } (rear + 1) \bmod MaxSize = front \\ \text{当队列为空: } front = rear \end{array} \right.$$

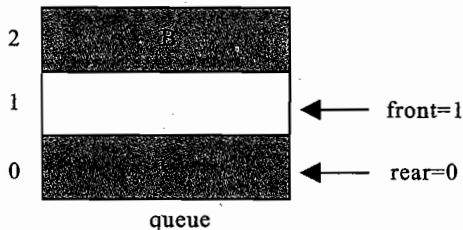
这两个条件所代表的意义是不一样的。例如有一队列 $queue$ 其 $MaxSize$ 为 3，其数据内容如下图所示：



若欲继续输入“C”，则判断队列是否已满

$$(rear + 1) \bmod \text{MaxSize} = (2+1) \bmod 3 = 0 \rightarrow \text{不等于 front}$$

则将“C”存入数组索引值为3的数组元素位置

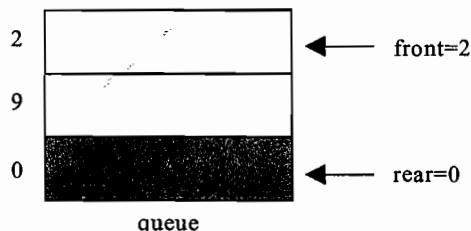


若欲继续输入“D”，也要判断队列是否已满

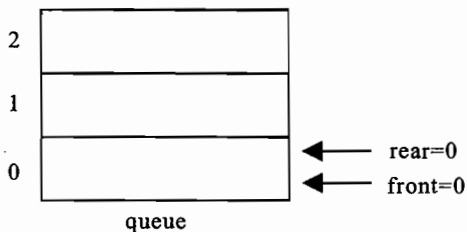
$$(rear + 1) \bmod \text{MaxSize} = (0+1) \bmod 3 = 1 \rightarrow \text{等于 front}$$

此时表示队列已满，故无法将“D”存入。

另外，若从环状队列输出数据时，也需要对指针做判断。例如有一队列 queue 其 MaxSize 为 3，其数据内容如下图所示：



若欲输出数据，则要判断队列是否已空。front 等于 2，不等于 rear，故输出 front 所指的下一个元素“A”，得到如下图：



欲再输出数据，此时 front 等于 rear，表示队列已满，无法再输出数据。

程序实例：

利用数组结构建立环状队列

程序构思：

环状队列需考虑判段队列为空或满的条件：

当队列为满时的条件为： $(rear + 1) \bmod \text{MaxSize}$

当队列空时条件为: $\text{front} = \text{rear}$

利用这两个条件来控制队列前后端数据的存取。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Queue_03.c */
03  /* 程序目的: 利用数组结构建立环状队列 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  # define MaxSize 10          /*队列的最大容量*/
08  int queue[MaxSize];          /*声明队列数组 */
09  int front = -1;              /*初始队头指针 */
10  int rear = -1;              /*初始队尾指针 */
11  /*-----*/
12  /*      输入队列数据      */
13  /*-----*/
14  void addqueue (int value)
15  {
16      rear=(rear+1) % MaxSize; /*用来控制尾指针的索引值*/
17      if ( front==rear)        /*检查队列是否已满 */
18          printf("The queue is full!!"); /*无法存入 */
19      else
20          queue[rear] = value; /*将数据存入队列*/
21  }
22  /*-----*/
23  /*      输出队列数据      */
24  /*-----*/
25  int delqueue ()
26  {
27      int temp;
28
29      if ( front == rear)      /*检查队列是否已空 */
30          return -1;
31      temp=queue[front+1];
32      queue[front]=0;
33      front=(front+1) % MaxSize; /*用来控制尾指针的索引值*/
34      return temp;            /*输出队列数据 */
35  }
36  /*-----*/
37  /*进行环状队列数据之输出输入, 并打输出队列内容 */
38  /*-----*/
39  /
40  void main ( )
41  {
42      int select ;            /*功能选项变量 */
43      int i,temp ;
44
45      for (i=0;i<=MaxSize;i++) /*清除数组内容*/
46          queue[i]=0;
47      printf("(1)Input a data\n");
48      printf("(2)Output a data\n");
49      printf("(3)Exit\n");
50      printf("Please select:");
51      scanf ( "%d", &select) ;
52      printf("\n");
53      if (select!=3)
54      {
55          do

```

```

56     {
57         switch ( select )
58         {
59             case 1 : printf ( "Please input the input value == > " ) ;
60                     scanf ("%d" , &temp) ;           /*读取输入值*/
61                     addqueue(temp);                 /*将数据插入队列中*/
62                     printf("The content of queue:"); /*输出队列内容*/
63                     for (i=0;i<=MaxSize;i++)
64                     {
65                         if (queue[i]!=0)
66                             printf("[%d]",queue[i]);
67                     }
68                     break;
69             case 2 : if ( (temp = delqueue () ) == -1 ) /*判断队列是否为空*/
70                     printf ("The queue is empty!!! \n ");
71                     else
72                     {
73                         printf ( "The output value is [%d]\n" , temp );
74                         printf("The content of queue:"); /*输出队列内容*/
75                         for (i=0;i<=MaxSize;i++)
76                         {
77                             if (queue[i]!=0)
78                                 printf("[%d]",queue[i]);
79                         }
80                     }
81                     break ;
82         }
83         printf("\n(1)Input a data\n");
84         printf("(2)Output a data\n");
85         printf("(3)Exit\n");
86         printf("Please select:");
87         scanf ( "%d", &select) ;
88         printf("\n");
89     } while (select !=3);
90 }
91 }

```

运行结果:

```

C:\DS>Queue_03
(1)Input a data
(2)Output a data
(3)Exit
Please select:1
Please input the input value==> 1

The content of queue: [1]

(1)Input a data
(2)Output a data
(3)Exit
Please select:2
The output value is [1]
The content of queue:

(1)Input a data
(2)Output a data
(3)Exit
Please select:2

```

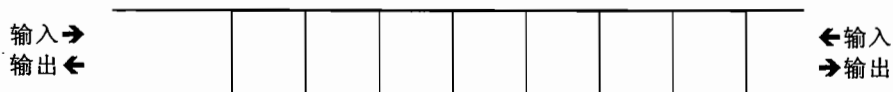
```
The queue is empty!!
```

```
(1)Input a data
(2)Output a data
(3)Exit
Please select:3
```

```
C:\DS>
```

5.5 双向队列

前面所提到的队列为单向队列，即从队列后端进行输入、前端进行输出。顾名思义，双向队列即可从前后端进行队列数据的输出及输入，如下图所示：



这样的结构最常用于计算机的 CPU 调度。所谓“CPU 调度”是在多人使用一个 CPU 的情况下，由于 CPU 在同一时间只能执行一项工作，故将每个人欲处理的工作先存于队列中，待 CPU 闲置时再从队列中取出一项待执行的工作进行处理。而在双向队列两端均可输出、输入的结构下，正好符合在 CPU 调度处理上的不同请求。

双向队列的设计有很多种，在此我们根据输出、输入方向的限制，将其分为两种：

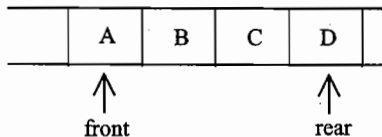
1. 输入限制性双向队列
2. 输出限制性双向队列

5.5.1 输入限制性双向队列

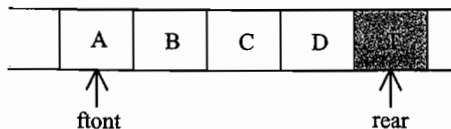
“输入限制性双向队列”是限制只能在队列的一端(后端)进行输入，而数据的输出则可在队列的两端(前后端)操作，如此一来会有两种情况：

1. 队列 [后端输入—前端输出]

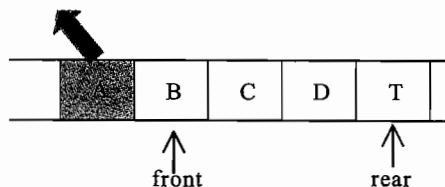
例如：



(a) 输入“T”

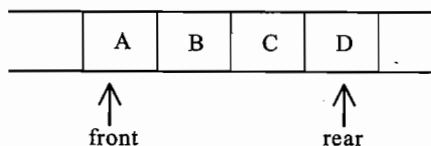


(b) 输出

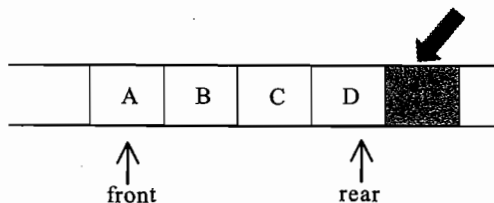


2. 队列 [后端输入—后端输出]

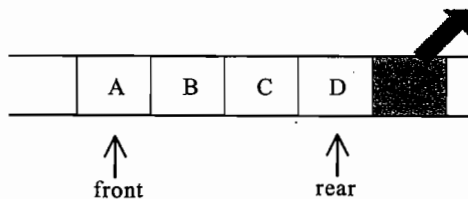
例如：



(a) 输入“T”



(b) 输出



程序实例：

已知一队列，欲使用“输入限制性双向队列”方式进行数据输出。

程序构思：

限制只能在队列的一端(后端)进行输入，而数据的输出则可在队列的两端(前后端)操作，可能有两种状况：“后端输入—前端输出”和“后端输入—后端输出”，分别考虑指针上的控制以达到输出输入的要求。



程序源代码：

```
01  /* ===== Program Description ===== */
02  /* 程序名称: Queue_04.c */
03  /* 程序目的: 输入限制性双向队列 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
```

```

07     # define MaxSize 10                                /*队列的最大容量*/
08     int queue[ MaxSize];                                /*声明队列数组 */
09     int front = -1;                                     /*初始队头指针 */
10     int rear = -1;                                     /*初始队尾指针 */
11     /*-----*/
12     /*          输入队列数据          */
13     /*-----*/
14     void addqueue (int value)
15     {
16         rear=(rear+1) % MaxSize;                        /*用来控制尾指针的索引值*/
17         if ( front==rear)                               /*检查队列是否已满 */
18             printf("The queue is full!!");
19         else
20             queue[rear] = value;                        /*将数据存入队列*/
21     }
22     /*-----*/
23     /*          输出队列数据(从后端)          */
24     /*-----*/
25     int rear_delqueue ()
26     {
27         int temp;
28
29         if (front == rear)                               /*检查队列是否为空 */
30             return -1;
31         temp=queue[rear];                                /*暂存输出数据*/
32         rear--;
33         if (rear < 0 && front !=-1)
34             rear=MaxSize - 1;
35         return temp;                                    /*输出队列数据*/
36     }
37     /*-----*/
38     /*          输出队列数据(从前端)          */
39     /*-----*/
40     int front_delqueue()
41     {
42         if (front == rear)                               /*检查队列是否为空 */
43             return -1;
44         front++;
45         if (front == MaxSize)                             /*超过最大指针时,再从头开始*/
46             front =0;
47         return queue[front];                             /*输出队列数据*/
48     }
49     /*-----*/
50     /*主程序:已知一个内含数据的队列,选择输出类型将数据输出, */
51     /*          并输出输出结果          */
52     /*-----*/
53     void main ( )
54     {
55         int select;                                     /*功能选择项*/
56         int output_queue[5];                           /*输出元素*/
57
58         int input_queue[5]={5,4,3,2,1};
59         int temp,position=0,i;
60
61         for (i=0;i<5;i++)                               /*将数组元素存入队列*/
62             addqueue(input_queue[i]);
63
64         printf("The original queue order :");
65         for (i=0;i<5;i++)
66             printf("[%d]",input_queue[i]);
67         printf("\n");

```

```

68     while (front != rear)
69     {
70         printf("(1)From 'Front-end'\n");
71         printf("(2)From 'Rear-end'\n");
72         printf("Please select one=>");
73         scanf("%d",&select);
74
75         switch (select)
76         {
77             case 1 :      /*从前端输出数据*/
78                 temp=front_delqueue();
79                 output_queue[position++]=temp;
80                 break;
81             case 2 :/*从后端输出数据 */
82                 temp=rear_delqueue();
83                 output_queue[position++]=temp;
84                 break;
85         }
86     }
87
88     printf("\nThe output order : ");
89     for (i=0;i<5;i++)
90         printf("[%d]",output_queue[i]);
91     printf("\n");
92 }

```

运行结果:

```

C:\DS>Queue_04
The original queue order : [5] [4] [3] [2] [1]

(1)From 'Front-end' out
(2)From 'Rear-end' out
Please select one=>2

(1)From 'Front-end' out
(2)From 'Rear-end' out
Please select one=>1

(1)From 'Front-end' out
(2)From 'Rear-end' out
Please select one=>1

(1)From 'Front-end' out
(2)From 'Rear-end' out
Please select one=>2

(1)From 'Front-end' out
(2)From 'Rear-end' out
Please select one=>2

The output order : [1] [5] [4] [2] [3]

C:\DS>

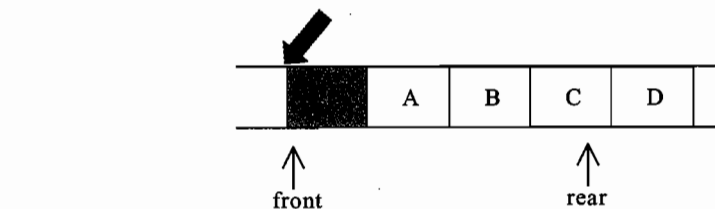
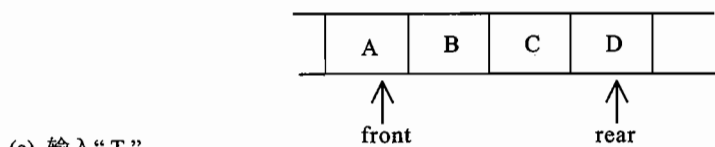
```


5.5.2 输出限制性双向队列

“输出限制性双向队列”恰好与“输入限制性双向队列”相反，它所限制的是只能在队列的一端(前端)进行输出，而数据的输入则是可以在队列的两端(前后端)操作。所以也会有两种情况：

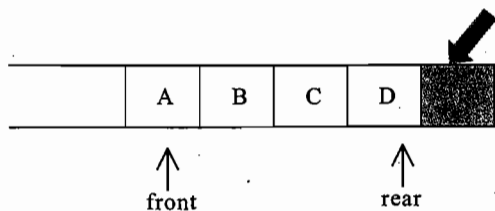
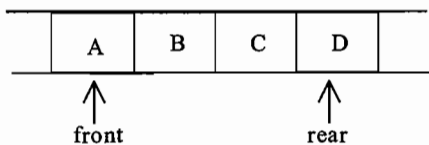
1. 队列 [前端输入—前端输出]

例如：

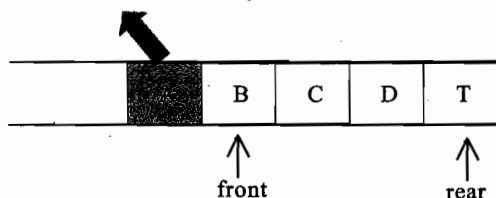


2. 队列 [后端输入—前端输出]

例如：



(b) 输出

**程序实例:**

已知一队列，欲使用“输出限制性双向队列”方式进行数据输出。

程序构思:

因为限制只能在队列的一端(前端)进行输出，而数据的输入则可在队列的两端(前后端)操作，可能有两种状况：“前端输入—前端输出”和“后端输入—前端输出”，分别考虑指针的控制以达到输出输入的要求。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Queue_05.c */
03  /* 程序目的: 输出限制性双向队列 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07  struct queue_node /*声明队列链接结构*/
08  {
09      int data;
10      struct queue_node *next;
11  };
12  typedef struct queue_node queue_list; /*定义队列类型 */
13  typedef queue_list *link; /*定义队列指针类型 */
14  link front = NULL; /*初始队头指针 */
15  link rear = NULL; /*初始队尾指针 */
16  /*-----*/
17  /* 输入队列数据 (从后端) */
18  /*-----*/
19  void rear_addqueue (int value)
20  {
21      link newnode;
22
23      newnode = ( link ) malloc (sizeof(queue_list) );
24
25      newnode->data = value; /*将数据存入队列*/
26      newnode->next = NULL; /*设置初值 */
27      if (rear==NULL) /*若为队列的第一个数据 */
28          front=newnode; /*将 front 指向新节点 */
29      else
30          rear->next =newnode; /*rear 所指的节点指向新节点 */
31          rear=newnode; /*rear 指向新节点 */
32  }
33  /*-----*/
34  /* 输入队列数据 (从后端) */
35  /*-----*/
36  void front_addqueue (int value)
37  {

```

```

38     link newnode;
39
40     newnode = ( link ) malloc (sizeof(queue_list) ) ;
41
42     newnode->data = value;           /*将数据存入队列*/
43     newnode->next= NULL;             /*设置初值 */
44     if (front==NULL)                /*若为队列的第一个数据 */
45     {
46         newnode->next=NULL;
47         rear=newnode;               /*rear 指向新节点 */
48     }
49     else
50         newnode->next=front; /*置入 front 的前面*/
51 }
52 /*-----*/
53 /* 输出队列数据 */
54 /*-----*/
55 int delqueue ()
56 {
57     link top;
58     int temp;
59
60     if ( front !=NULL)              /*检查队列是否已空 */
61     {
62         top = front ;               /*将top 指向 front */
63         front = front -> next;      /*删除之前节点*/
64         temp = top -> data ;         /*暂存输出队列数据 */
65         free (top);                 /*释放输出节点的内存*/
66         return temp;                /*输出队列数据*/
67     }
68     else
69         return -1 ;
70 }
71 /*-----*/
72 /*主程序:已知一个内含数据的队列,选择输出类型将数据输出, */
73 /*      并显示输出结果 */
74 /*-----*/
75 void main ( )
76 {
77     int select ;                    /*功能选项变量 */
78     int i,temp,position ;
79     int input_queue[6]={6,5,4,3,2,1};
80
81     printf("The original queue order :");
82     for (i=0;i<6;i++)
83         printf("[%d]",input_queue[i]);
84
85     for (i=0;i<6;i++)
86     {
87         printf("\n(1)From 'Front-end' in\n");
88         printf("(2)From 'Rear-end' in\n");
89         printf("Please select one=>");
90         scanf("%d",&select);
91         switch (select)
92         {
93             case 1 : /*从前端输入数据*/
94                 front_addqueue(input_queue[i]);
95                 break;
96             case 2 : /*从后端输出数据 */
97                 rear_addqueue(input_queue[i]);
98                 break;

```

```

99         default: /*默认为后端输入*/
100             rear_addqueue(input_queue[i]);
101             break;
102         }
103     }
104     printf("\nThe output order : ");
105     while ( (temp=delqueue ()) != -1)
106         printf("[%d]",temp);
107     printf("\n");
108 }

```

运行结果:

```

C:\DS>Queue_04
The original queue order : [6] [5] [4] [3] [2] [1]

(1)From 'Rear-end' in
(2)From 'Front-end' in
Please select one=>1

(1)From 'Rear-end' in
(2)From 'Front-end' in
Please select one=>2

(1)From 'Rear-end' in
(2)From 'Front-end' in
Please select one=>1

(1)From 'Rear-end' in
(2)From 'Front-end' in
Please select one=>2

(1)From 'Rear-end' in
(2)From 'Front-end' in
Please select one=>1

1)From 'Rear-end' in
(2)From 'Front-end' in
Please select one=>2

The output order : [1] [3] [5] [6] [4] [2]

C:\DS>

```

【习题】

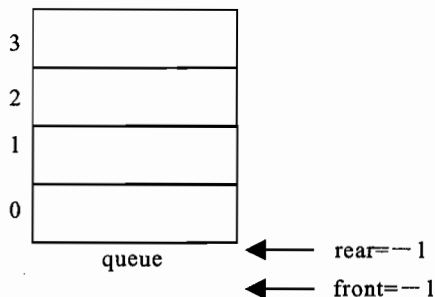
一、复习:

1. 请说明何谓“队列”? 并举出3个队列的应用。
2. 队列的建立可使用两种结构: _____结构和_____结构。
3. 队列的数据存取特性为_____。
4. 若将5,4,3,2,1依序存入队列中,则将数据从队列取出直到队列为空的顺序为_____。
5. 试说明为何要使用“环状队列”?

二、应用:

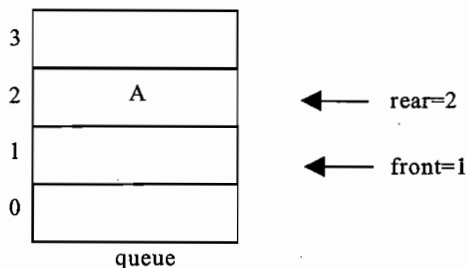
1. 试比较数组队列和链表队列的差异。

2. 试说明“堆栈”和“队列”的异同。
3. 请说明下列两种双向队列的差别。
 - (1) 输入限制性双向队列
 - (2) 输出限制性双向队列
4. 请写出执行下列函数之后的结果：(此为一队列)
 - addqueue() 结果包含“队列数据内容”、“front”及“rear”
 - delqueue() 结果包含“队列数据内容”、“front”、“rear”及输出数据值



- (1) addqueue (A) (2) addqueue (B) (3) delqueue ()
 (4) addqueue (C) (5) delqueue () (6) delqueue ()

5. 请写出执行下列函数之后的结果：(此为一环状队列)
 - addcqueue() 结果包含“队列数据内容”、“front”及“rear”
 - delcqueue() 结果包含“队列数据内容”、“front”、“rear”及输出数据值



- (1) addcqueue(B) (2) addcqueue(C) (3) addcqueue(D)
 (4) delcqueue() (5) addcqueue(E) (6) delcqueue()

6. 请使用链表结构写出下列程序：
 - (1) “输入限制性双向队列”
 - (2) “输出限制性双向队列”



递 归

第 6 章

- ◆ 何谓递归
- ◆ 函数调用与参数传递
- ◆ 数学问题
- ◆ 河内塔问题
- ◆ N 皇后问题
- ◆ 迷宫问题

6.1 何谓递归

递归(Recursive)简单的定义就是子程序或函数重复的调用自己,并传入不同的变量来执行的一种程序设计技巧,而递归在程序设计及解题上也是一种有力、重要的工具,帮助程序设计者解决复杂的问题,并精简程序结构。

比如说,我们现在想设计一个乘法器,这个乘法器可计算出两数相乘的乘积,但是此时计算机并不提供乘法运算(我们仅知道任何数乘 1,其值不变),我们唯一可以使用运算方式是加法运算,那么我们该如何来设计这个程序呢?

回想一下,在日常生活中,如果我们做两数相乘,但仅能使用加法,不能使用乘法,此时我们该如何解题的呢?思考一下,13*4 的答案是如何算出的,13*4 是不是就是 13 连加 4 次呢?那我们如何得知 13 还要连加几次?何时停止连加呢?

程序实例:

设计一个可计算两数相乘,但仅用加法运算,不使用乘法运算的程序。

程序构思:

假设欲计算出 13*4,则:

$$\begin{aligned}13 * 4 &= 13 + (13 * 3) \\&= 13 + (13 + (13 * 2)) \\&= 13 + (13 + (13 + (13 * 1))) \\&= 13 + (13 + (13 + 13)) \\&= 13 + (13 + 26) \\&= 13 + 39 \\&= 52\end{aligned}$$

由以上的运算过程我们可以明显的看出,每一次运算都有一定的规则,在上例中,每次运算时 13 所乘的数依序递减,最后再返回运算结果与前数相加。

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: multiply.c */
03  /* 程序目的: 设计一个可计算两数相乘,但仅用加法运算, */
04  /*           不使用乘法运算的程序。 */
05  /* Written By Kuo-Yu Huang. (WANT Studio.) */
06  /* ===== */
07
08  /* ----- */
09  /* 递归乘法运算 */
10  /* ----- */
11  int Multiply(int M,int N)
12  {
13      int Result; /*运算结果*/
14
15      if ( N == 1)
16          Result = M; /* 递归结束条件 */
17      Else
18          Result = M + Multiply(M,N-1); /* 递归执行部分 */
19      return Result;
20  }
21
22
23  /* ----- */
```

```

24  /* 主程序 */
25  /* ----- */
26  void main ()
27  {
28      int    NumA;          /* 乘数变量 */
29      int    NumB;          /* 被乘数变量 */
30      int    Product;       /* 乘积变量 */
31
32      printf("Please enter Number A:"); /* 输入乘数 */
33      scanf("%d",&NumA);
34      printf("Please enter Number B:"); /* 输入被乘数 */
35      scanf("%d",&NumB);
36
37      Product = multiply(NumA,NumB);
38      printf("%d * %d = %d",NumA,NumB,Product);
39  }

```

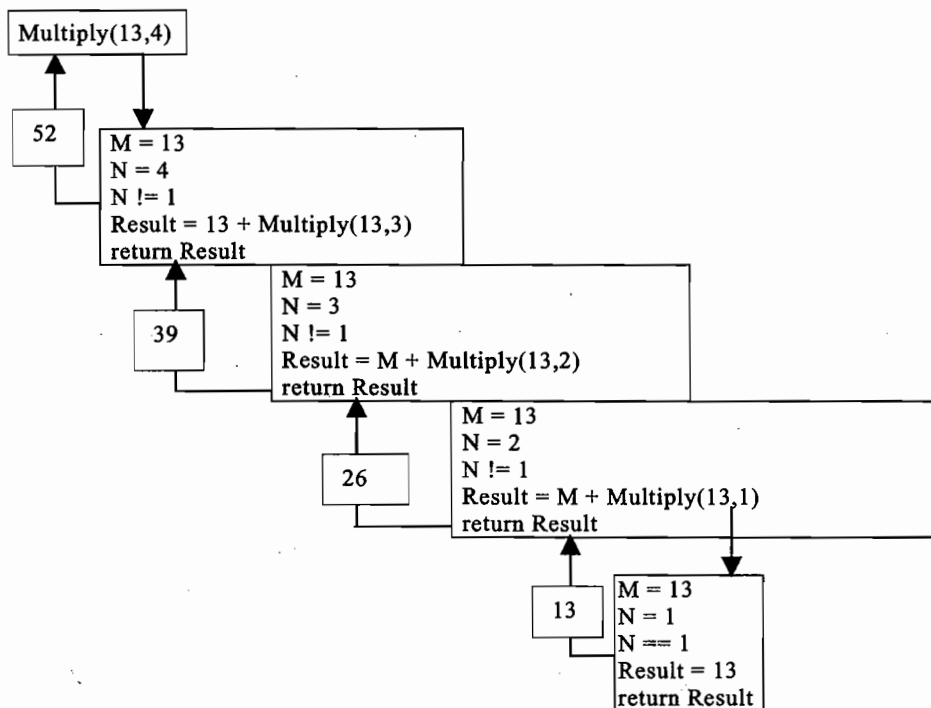
运行结果:

```

C:\DS>multiply
Please enter Number A:13
Please enter Number B:4
13 * 4 = 52
C:\DS>

```

程序流程:



从以上的实例中, 我们可归纳出几个解递归问题的步骤:

步骤 1: 了解题意是否适合用递归来解题。

步骤 2: 决定递归结束条件(Stopping Cases)。

步骤 3: 决定递归执行部分(Recursive Step)。

我们由题意可知每次执行的过程相似, 唯一的不同点为其中一个传入参数, 每次执行都递减。递归

结束条件为当被乘数为 1 时返回乘数的值。否则继续调用程序并递减传入被乘数值。其结构如下：

```
int Multiply(int M,int N)
{
    int Result;
    if ( N == 1)
        Result = M;                /* 递归结束条件 (Stopping Case) */
    else
        Result = M + Multiply(M,N-1); /* 递归执行部分 (Recursive Step) */
    return Result;
}
```

说明

处理递归问题，常采用 if 语句来判断是否符合递归结束条件，其算法格式如下：

```
if (符合条件) then
    返回 答案
else
```

使用递归将程序分割为更简单的小程序。

6.2 函数调用与参数传递

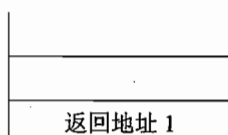
在 C 语言中，我们采用堆栈这个数据结构来记录函数调用后的返回地址。例如有一个程序如下：

```
int    ProcedureA ()          /*    子程序 A    */
{
    ...
    ProcedureB();             /*    调用子程序 B    */
    ...                       /*    返回地址 2    */
}

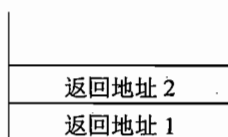
int    ProcedureB()           /*    子程序 B    */
{
    ...
}

void main ()                  /*    主程序    */
{
    ...
    ProcedureA();             /*    调用子程序 A    */
    ...                       /*    返回地址 1    */
}
```

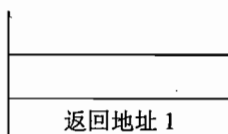
则当主程序执行到“ProcedureA()”(调用子程序 A)这行时，堆栈需记录下一行程序语句的地址，也就是将“返回地址 1”存入(PUSH)堆栈中，以便在 ProcedureA 执行完之后，能顺利返回主程序中继续执行未执行之程序语句。这时堆栈中的内容为：



调用 ProcedureA 后, 当 ProcedureA 执行到 “ProcedureB()” (调用子程序 B) 这行时, 堆栈仍需记录下一行程序语句的地址, 也就是将 “返回地址 2” 存入(PUSH)堆栈中, 以便在 ProcedureB 执行完之后, 能顺利返回 ProcedureA 中继续执行未执行的程序语句。这时堆栈中的内容为:

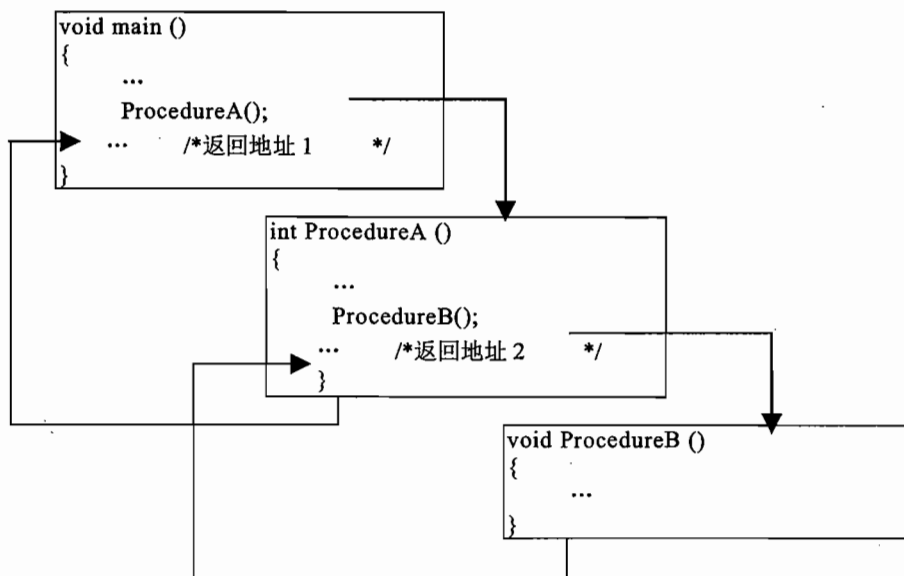


当 ProcedureB 执行完后, “程序返回地址 2” 便从堆栈中被取出(POP), 继续执行 ProcedureA 中未执行的程序语句。这时堆栈中的内容为:



当 ProcedureA 执行完后, “程序返回地址 1” 便从堆栈中被取出(POP), 继续执行主程序中未执行的程序语句。这时堆栈无任何的内容。

其程序执行的流程如下图:



对于递归而言, 也是反复的调用子程序的结构, 如同上列函数的调用, 递归也需要运用堆栈来记录程序的返回地址, 以下我们就利用一个例子来说明递归程序执行的流程。

程序实例:

运用递归设计一个将字符串反转的程序。

程序构思:

递归结束条件: 当字符串中每个字符都输出时。

递归执行部分: 从字符串到后一个字符开始输出, 依次输到字符串最前的字符。

需先知道字符串的内容及长度。

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: reverse.c */
03  /* 程序目的: 运用递归设计一个将字符串反转的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06
07  char String[30];          /* 声明字符串变量 */
08  int Length;              /* 字符串长度变量 */
09
10  /* ----- */
11  /* 递归字符串反转 */
12  /* ----- */
13  void Reverse(int N)
14  {
15      if ( N < Length)
16      {
17          Reverse(N+1);    /* 递归执行部分 */
18          printf("%c",String[N]);
19      }
20  }
21
22  /* ----- */
23  /* 主程序 */
24  /* ----- */
25  void main ()
26  {
27
28      printf("Please enter string : ");    /* 输入原字符串 */
29      scanf("%s",&String);
30
31      Length = strlen(String);            /* 取得字符串长度 */
32      printf("The reverse string : ");
33      Reverse(0);                        /* 调用递归函数 */
34      printf("\n");
35  }
```

运行结果:

```
C:\DS>reverse
Please enter string : ABC
The reverse string : CBA

C:\DS>
```

堆栈变化:

步骤 1: 则当主程序执行到“Reverse(0)”这行时, 堆栈需记录下一行程序语句的地址, 也就是将“第 34 行语句的地址”存入堆栈中。这时堆栈中的内容为:

第 34 行语句的地址

步骤 2: 进入 Reverse 子程序中执行到“Reverse(1)”这行时, 堆栈需记录下一行程序语句的地址, 也就是将“第 18 行语句的地址”存入堆栈中。这时堆栈中的内容为:

第 18 行语句的地址
第 34 行语句的地址

步骤 3: 进入 Reverse 子程序中执行到“Reverse(2)”这行时, 堆栈需记录下一行程序语句的地址, 也就是将“第 18 行语句的地址”存入堆栈中。这时堆栈中的内容为:

第 18 行语句的地址
第 18 行语句的地址
第 34 行语句的地址

步骤 4: 此时 N=3, N 值已经不小于字符串长度(Length), 故不继续执行递归部分, 而将“第 18 行语句的地址”从堆栈中取出, 继续执行第 18 行语句的程序语句, 输出字符串数组[2]的字符。这时堆栈中的内容为:

第 18 行语句的地址
第 34 行语句的地址

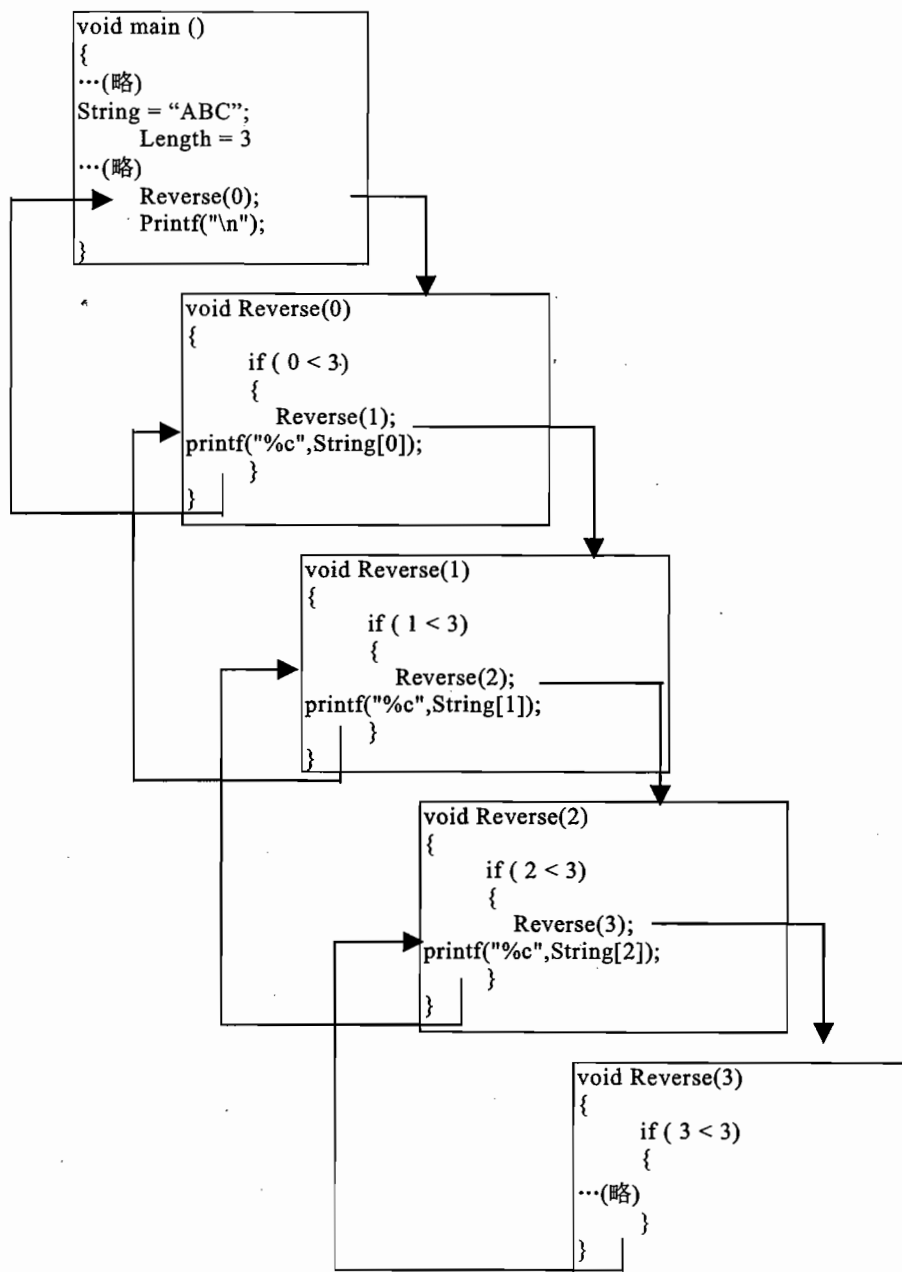
步骤 5: 将“第 18 行语句的地址”从堆栈中取出, 继续执行第 18 行语句的程序语句, 输出字符串数组[1]的字符。这时堆栈中的内容为:

第 34 行语句的地址

步骤 6: 将“第 18 行语句的地址”从堆栈中取出, 继续执行第 18 行语句的程序语句, 输出字符串数组[0]的字符。这时堆栈无任何的内容。

此时我们可在屏幕上看到程序执行后输出为“CBA”。

程序流程:



6.3 数学问题

本节所要讨论的是如何运用递归来处理一些常见的数学问题。

6.3.1 阶乘问题

数学上阶乘(Factorial)运算的定义为:

$$n! = \begin{cases} 1 & , n \leq 1 \\ n * (n-1)! & , n > 1 \end{cases}$$

我们可运用递归来设计阶乘的运算。

当 n 小于或等于 1, 返回阶乘为 1, 我们定义出 $0! = 1, 1! = 1$ 。

当 n 大于 1 时, 返回阶乘为 $n! = n * (n-1)!$ 。

例如欲求 $7!$, 则:

$$\begin{aligned} 7! &= 7 * 6! = 7 * (6 * 5!) \\ &= 7 * 6 * (5 * 4!) = 7 * 6 * 5 * (4 * 3!) \\ &= 7 * 6 * 5 * 4 * (3 * 2!) = 7 * 6 * 5 * 4 * 3 * (2 * 1!) \\ &= 7 * 6 * 5 * 4 * 3 * (2 * 1) = 7 * 6 * 5 * 4 * (3 * 2) \\ &= 7 * 6 * 5 * (4 * 6) = 7 * 6 * (5 * 24) \\ &= 7 * (6 * 120) = 7 * 720 \\ &= 5040 \end{aligned}$$

程序实例:

运用递归设计一个做阶乘运算的程序。

程序构思:

递归结束条件:

当阶乘数小于或等于 1 时, 返回 1。

递归执行部分:

当阶乘数大于 1 时, 返回 $n! = n * (n-1)!$ 。

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: factor.c */
03  /* 程序目的: 运用递归设计一个做阶乘运算的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06
07  /* ----- */
08  /* 递归阶层运算 */
09  /* ----- */
10  int Factor(int N)
11  {
12      if ( N <= 1)                /* 递归结束条件 */
13          return 1;
14      else
15          return N * Factor(N-1); /* 递归执行部分 */
16  }
17
18  /* ----- */
```

```

19  /* 主程序 */
20  /* ----- */
21  void main ()
22  {
23      int    Number;          /* 运算数值变量 */
24      int    Factorial;       /* 阶乘数值变量 */
25
26      printf("Please enter a number : "); /* 输入数值 */
27      scanf("%d",&Number);
28
29      Factorial = Factor(Number); /* 调用递归函数 */
30      printf("%d! = %d\n",Number,Factorial); /* 输出运算结果 */
31  }

```

运行结果:

```

C:\DS>factor
Please enter a number : 7
7! = 5040

C:\DS>

```

6.3.2 最大公因子问题

数学上求最大公因子(Great Common Divisor)的运算时常使用辗转相除法,反复计算到余数为零为止,这种方法也称为欧几里得定理(Euclid's Algorithm),其定义为:

$$\text{GCD}(M,N) = \begin{cases} M & , N = 0 \\ \text{GCD}(N, M \% N) & , N > 0 \end{cases}$$

我们可运用递归来设计求最大公因子的程序。

当 N 等于 0, 返回最大公因子为 M。

当 N 大于 0 时, 返回最大公因子为 GCD(N, M % N)。

例如欲求 GCD(3155,2545), 则:

$$\begin{aligned}
 \text{GCD}(3155,2545) &= \text{GCD}(2545,3155 \% 2545) = \text{GCD}(2545,610) \\
 &= \text{GCD}(610,2545 \% 610) = \text{GCD}(610,105) \\
 &= \text{GCD}(105,610 \% 105) = \text{GCD}(105,85) \\
 &= \text{GCD}(85,105 \% 85) = \text{GCD}(85,20) \\
 &= \text{GCD}(20,85 \% 20) = \text{GCD}(20,5) \\
 &= \text{GCD}(5,20 \% 5) = \text{GCD}(5,0) \\
 &= 5
 \end{aligned}$$

程序实例:

运用递归设计一个求两数之最大公因子的程序。

程序构思:

递归结束条件:

当 N = 0, 返回最大公因子为 M。

递归执行部分:

当 $N > 0$ 时, 返回最大公因子为 $GCD(N, M \% N)$ 。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: gcd.c */
03  /* 程序目的: 运用递归设计一个求两数之最大公因子的程序 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06
07  /* ----- */
08  /* 递归求最大公因子 */
09  /* ----- */
10  int GCD(int M, int N)
11  {
12      if (N == 0)          /* 递归结束条件 */
13          return M;
14      else
15          return GCD(N, M % N); /* 递归执行部分 */
16  }
17
18  /* ----- */
19  /* 主程序 */
20  /* ----- */
21  void main ()
22  {
23      int    NumberA;      /* 运算数值变量 */
24      int    NumberB;      /* 运算数值变量 */
25      int    Result;       /* 运算结果变量 */
26
27      printf("The Great Common Divisor of Number A, Number B\n");
28      printf("Please enter number A : "); /* 输入数值 */
29      scanf("%d", &NumberA);
30      printf("Please enter number B : "); /* 输入数值 */
31      scanf("%d", &NumberB);
32
33      Result = GCD(NumberA, NumberB); /* 调用递归函数 */
34      printf("GCD(%d,%d) = %d\n", NumberA, NumberB, Result);
35  }

```

运行结果:

```

C:\DS>gcd
The Great Common Divisor of Number A, Number B
Please enter number A : 2545
Please enter number B : 3155
GCD(2545,3155) = 5

C:\DS>

```

6.3.3 费氏级数问题

数学上有一种费氏级数(Fibonacci Numbers), 其定义为:

$$\text{Fib}(N) = \begin{cases} N & , N \leq 1 \\ \text{Fib}(N-1) + \text{Fib}(N-2) & , N > 1 \end{cases}$$

我们可运用递归来设计求费氏级数的程序。

当 N 小于或等于 1, 返回费氏级数值为 N 。

当 N 大于 1 时, 返回费氏级数值为 $\text{Fib}(N-1) + \text{Fib}(N-2)$ 。

例如欲求 $\text{Fib}(5)$, 则:

$$\begin{aligned}
 \text{Fib}(5) &= \text{Fib}(4) + \text{Fib}(3) \\
 &= \text{Fib}(3) + \text{Fib}(2) + \text{Fib}(2) + \text{Fib}(1) \\
 &= \text{Fib}(2) + \text{Fib}(1) + \text{Fib}(1) + \text{Fib}(1) + 1 \\
 &= \text{Fib}(1) + 1 + 1 + 1 + 1 \\
 &= 1 + 1 + 1 + 1 + 1 \\
 &= 2 + 1 + 1 + 1 \\
 &= 3 + 2 \\
 &= 5
 \end{aligned}$$

程序实例:

运用递归设计一个求费氏级数的程序。

程序构思:

递归结束条件:

当 $N \leq 1$, 返回费氏级数值为 N 。

递归执行部分:

当 $N > 1$ 时, 返回费氏级数值为 $\text{Fib}(N-1) + \text{Fib}(N-2)$ 。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: fib.c */
03  /* 程序目的: 运用递归设计一个求费氏级数的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06
07  /* ----- */
08  /* 递归求费氏级数 */
09  /* ----- */
10  int Fib(int N)
11  {
12      if (N <= 1) /* 递归结束条件 */
13          return N;
14      else
15          return Fib(N-1) + Fib(N-2); /* 递归执行部分 */
16  }
17
18  /* ----- */
19  /* 主程序 */
20  /* ----- */
21  void main ()
22  {
23      int    Number; /* 运算数值变量 */
24      int    Result; /* 运算结果变量 */
25
26      printf("The Fibonacci Numbers \n");
27      printf("Please enter a number : "); /* 输入数值 */

```

```
28     scanf("%d",&Number);
29
30     Result = Fib(Number);          /* 调用递归函数 */
31     printf("Fibonacci Numbers of %d = %d\n",Number,Result);
32 }
```

运行结果:

```
C:\DS>fib
The Fibonacci Numbers
Please enter a number : 10
Fibonacci Numbers of 10 = 55

C:\DS>
```

6.3.4 组合公式

数学上组合公式的求法，其定义为：

$$C_n^m = \begin{cases} 1, & (n=m) \text{ or } (m=0) \\ C_n^{m-1} + C_{n-1}^{m-1}, & \text{otherwise} \end{cases}$$

我们可运用递归来设计求组合公式的程序。

当 n 等于 m 或 $m=0$ 时，返回 1。

否则返回 $C_n^{m-1} + C_{n-1}^{m-1}$ 。

例如欲求 $\text{Comb}(5,3)$ ，则：

$$\begin{aligned} \text{Comb}(5,3) &= \text{Comb}(4,3) + \text{Comb}(4,2) \\ &= \text{Comb}(3,3) + \text{Comb}(3,2) + \text{Comb}(3,2) + \text{Comb}(3,1) \\ &= 1 + \text{Comb}(2,2) + \text{Comb}(2,1) + \text{Comb}(2,2) + \text{Comb}(2,1) + \\ &\quad \text{Comb}(2,1) + \text{Comb}(2,0) \\ &= 1 + 1 + \text{Comb}(1,1) + \text{Comb}(1,0) + 1 + \text{Comb}(1,1) + \text{Comb}(1,0) \\ &\quad + \text{Comb}(1,1) + \text{Comb}(1,0) + 1 \\ &= 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ &= 1 + 1 + 2 + 1 + 2 + 2 + 1 \\ &= 1 + 3 + 3 + 3 \\ &= 4 + 6 \\ &= 10 \end{aligned}$$

程序实例：

运用递归设计一个求组合公式的程序。

程序构思：

递归结束条件：

当 n 等于 m 或 $m=0$ 时，返回 1。

递归执行部分:

否则返回 $\text{Comb}(n-1,m) + \text{Comb}(n-1,m-1)$ 。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: comb.c */
03  /* 程序目的: 运用递归设计一个求组合公式的程序。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06
07  /* ----- */
08  /* 递归求组合公式 */
09  /* ----- */
10  int Comb(int N,int M)
11  {
12      if ( (N == M) || (M == 0) ) /* 递归结束条件 */
13          return 1;
14      else /* 递归执行部分 */
15          return Comb(N-1,M) + Comb(N-1,M-1);
16  }
17
18  /* ----- */
19  /* 主程序 */
20  /* ----- */
21  void main ()
22  {
23      int    NumberN; /* 运算数值变量 */
24      int    NumberM; /* 运算数值变量 */
25      int    Result; /* 运算结果变量 */
26
27      printf("The Combination Number of two Numbers.\n");
28      printf("Please enter number N: "); /* 输入数值 */
29      scanf("%d",&NumberN);
30      printf("Please enter number M: "); /* 输入数值 */
31      scanf("%d",&NumberM);
32
33      if (NumberN >= NumberM)
34      {
35          Result = Comb(NumberN,NumberM); /* 调用递归函数 */
36          printf("Comb(%d,%d) = %d\n",NumberN,NumberM,Result);
37      }
38      else
39          printf("Error: N < M !!\n");
40  }

```

运行结果:

```

C:\DS>comb
The Combination Number of two Numbers.
Please enter number N: 9
Please enter number M: 3
Comb(9,3) = 84

C:\DS>comb
The Combination Number of two Numbers.
Please enter number N: 3
Please enter number M: 9
Error: N < M !!

```

C:\DS>

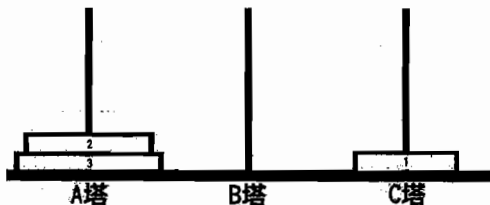
6.4 汉诺塔问题

相传在某一座古庙有 3 根木桩，有 24 个铁盘由小到大的放置在其中 1 根木桩上，庙中流传着一个传说：“如果有一天能把 24 个铁盘，从其中一根木桩移至另一根木桩，且必须遵守着：(1)每天只能搬动 1 个铁盘，而且只能从最上面的铁盘开始搬动。(2)必须维持较小的铁盘在上方的原则。这两个原则，则当 24 个铁盘完全搬至另一个木桩时，世界就会永久和平”。这个问题就是著名的汉诺塔(Tower of Hanoi)问题。

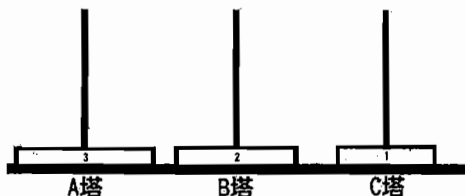
以下我们便实际操作看看当只有 3 个铁盘时的操作步骤，进而来思考这个汉诺塔问题该如何解？是否有其规则可寻，并计算出需花多少时间来搬动铁盘，证实这个传说。



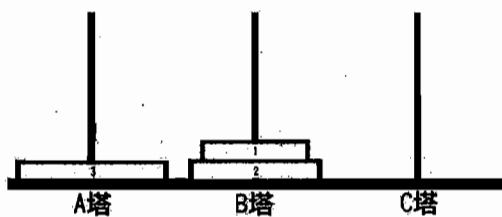
步骤 1：将 1 号铁盘从 A 桩搬至 C 桩。



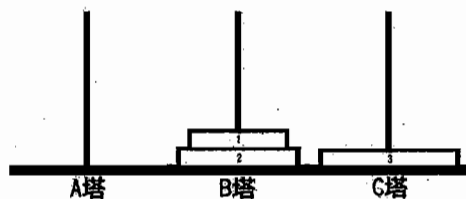
步骤 2：将 2 号铁盘从 A 桩搬至 B 桩。



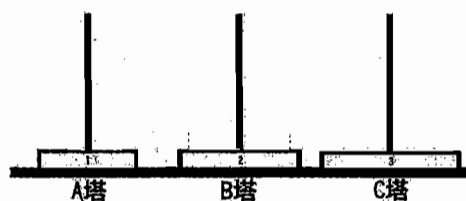
步骤 3：将 1 号铁盘从 C 桩搬至 B 桩。



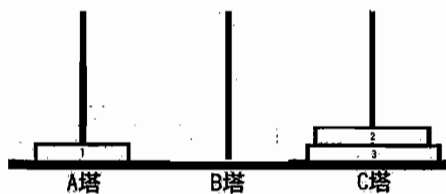
步骤 4：将 3 号铁盘从 A 桩搬至 C 桩。



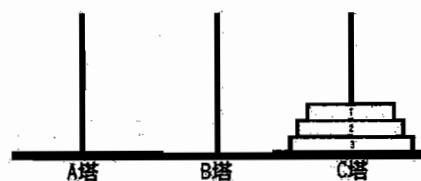
步骤 5：将 1 号铁盘从 B 桩搬至 A 桩。



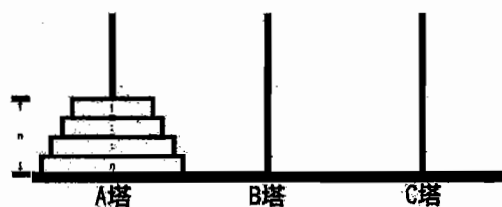
步骤 6：将 2 号铁盘从 B 桩搬至 C 桩。



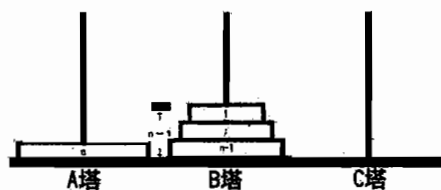
步骤 7: 将 1 号铁盘从 A 桩搬至 C 桩。



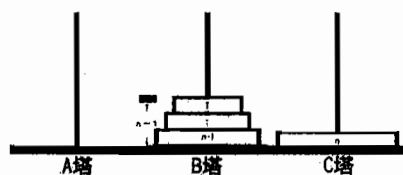
从处理 3 个铁盘的汉诺塔问题, 我们可发现当铁盘的数量愈多时, 铁盘搬动的次数也会愈多, 但可归纳出一些规则:



步骤 1: 将前 $n-1$ 号铁盘从 A 桩搬至 B 桩。



步骤 2: 将 n 号铁盘从 A 桩搬至 C 桩。



步骤 3: 将前 $n-1$ 号铁盘从 B 桩搬至 C 桩。



程序实例:

运用递归来解汉诺塔问题。

程序构思:

由上述对 n 个铁盘的汉诺塔问题分析中, 我们可定义出铁盘原先所在的桩为“来源桩”(Source), 铁盘欲移往的桩为“目的桩”(Destination), 而另一个桩为“辅助桩”(Auxiliary)。

当未移往目的桩的铁盘数为 1 时, 则将最后所剩的铁盘移至目的桩, 即完成工作。

否则, (1)将前 $N-1$ 个铁盘从来源桩移往辅助桩。(2)将编号为 N 的铁盘从来源桩移往目的桩。(3)将前 $N-1$ 个铁盘从辅助桩移往目的桩。

递归结束条件:

当未移往目的桩的铁盘数为 1 时, 将编号为 1 的铁盘移至目的桩。

递归执行部分:

- (1) 将前 $N-1$ 个铁盘从来源桩移往辅助桩。
- (2) 将编号为 N 的铁盘从来源桩移往目的桩。
- (3) 将前 $N-1$ 个铁盘从辅助桩移往目的桩。

程序源代码:

```
01  /* ===== Program Description ===== */
02  /* 程序名称: hanoi.c */
03  /* 程序目的: 运用递归来解汉诺塔问题。 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06  #include <stdio.h>
07  int Counter;          /* 计数器变量 */
08
09  /* ----- */
10  /* 递归解汉诺塔问题 */
11  /* ----- */
12  int Hanoi(char From, char To, char Auxiliary, int N)
13  {
14      if ( N == 1 )      /* 递归结束条件 */
15      {
16          Counter++; /* 计数器递增 */
17          printf("Step %d : ", Counter);
18          printf("Move disk 1 from peg-%c to peg-%c.\n", From, To);
19      }
```

```

20     else                /* 递归执行部分 */
21     {
22         /* 将目的桩和辅助桩交换 */
23         Hanoi(From,Auxiliary,To,N-1);
24         Counter++; /* 计数器递增 */
25         printf("Step %d : ",Counter);
26         printf("Move disk %d from peg-%c to peg-%c.\n",N,From,To);
27         /* 将来源桩和辅助桩交换 */
28         Hanoi(Auxiliary,To,From,N-1);
29     }
30 }
31
32 /* ----- */
33 /* 主程序 */
34 /* ----- */
35 void main ()
36 {
37     int    Number;        /* 铁盘数目变量 */
38     char   Source;
39     char   Destination;
40     char   Auxiliary;
41
42     Counter = 0;
43     printf("The Tower of Hanoi program.\n");
44     printf("Please enter the number of disks : ");
45     scanf("%d",&Number); /* 输入铁盘数 */
46     printf("The Source peg : ");
47     Source = getche();    /* 输入来源桩 */
48     printf("\nThe Auxiliary : ");
49     Auxiliary = getche(); /* 输入辅助桩 */
50     printf("\nThe Destination : ");
51     Destination = getche(); /* 输入目的桩 */
52     printf("\n");
53
54     Hanoi(Source,Destination,Auxiliary,Number); /* 调用递归函数 */
55 }
56

```

运行结果:

```

C:\DS>hanoi
The Tower of Hanoi program.
Please enter the number of disks : 4
The Source peg : A
The Auxiliary : B
The Destination : C
Step 1 : Move disk 1 from peg-A to peg-B.
Step 2 : Move disk 2 from peg-A to peg-C.
Step 3 : Move disk 1 from peg-B to peg-C.
Step 4 : Move disk 3 from peg-A to peg-B.
Step 5 : Move disk 1 from peg-C to peg-A.
Step 6 : Move disk 2 from peg-C to peg-B.
Step 7 : Move disk 1 from peg-A to peg-B.
Step 8 : Move disk 4 from peg-A to peg-C.
Step 9 : Move disk 1 from peg-B to peg-C.
Step 10 : Move disk 2 from peg-B to peg-A.
Step 11 : Move disk 1 from peg-C to peg-A.
Step 12 : Move disk 3 from peg-B to peg-C.
Step 13 : Move disk 1 from peg-A to peg-B.
Step 14 : Move disk 2 from peg-A to peg-C.

```



```
Step 15 : Move disk 1 from peg-B to peg-C.
```

```
C:\DS>
```

完成了这个汉诺塔程序，读者可自行利用这个程序来解解传说中的二十四个铁盘需要多久才能搬完。(注意：本程序中计数器声明为整数类型，只允许最大数为 32767，故解二十四个铁盘的河内塔问题需做些修改。)

从本节所归纳出的规则，我们得到：

当未移往目的桩的铁盘数为 1 时，则将最后所剩的铁盘移至目的桩，即完成工作。此时搬动的次数为一次。

否则，(1)将前 N-1 个铁盘从来源桩移往辅助桩。(2)将编号为 N 的铁盘从来源桩移往目的桩。(3)将前 N-1 个铁盘从辅助桩移往目的桩。此时搬动的次数为 N-1 个铁盘搬动的次数加上最后一个铁盘需搬动一次。

$$T(n) = \begin{cases} 1 & , n=1 \\ T(n-1)+1 & , n>1 \end{cases}$$

经过计算后可得 $T(n) = 2^n - 1$ ，所以 24 个铁盘要花上 $2^{24} - 1$ 天来搬动。(关于递归关系式的计算，可参第 1 章时间复杂度计算。)


6.5 N 皇后问题

在国际象棋中，皇后的势力范围包括上、下、左、右、左上、左下、右上、右下八个方向。N 皇后问题，就是求在一个 $N \times N$ 的棋盘上放置 N 个皇后的方法解。在解 N 皇后问题之前，我们先来讨论一下在 4×4 的棋盘上该如何放置 4 个皇后，从中探讨出解决 N 皇后问题的规则。假设有一个 4×4 的棋盘如下：

	0	1	2	3
0				
1				
2				
3				

步骤 1:

(0,0) → 放置成功





	0	1	2	3
0				
1				
2				
3				

步骤 2:

(1,0) → 放置失败(∵第1列已有1个皇后)

(1,1) → 放置失败(∵左上角已有1个皇后)

(1,2) → 放置成功

	0	1	2	3
0				
1				
2				
3				







步骤 3:

(2,0) → 放置失败(∵第1列已有1个皇后)

(2,1) → 放置失败(∵右上角已有1个皇后)

(2,2) → 放置失败(∵第3列已有1个皇后)

(2,3) → 放置失败(∵左上角已有1个皇后)



	0	1	2	3
0				
1				
2				
3				

步骤 4:

第3行无法放置任何皇后, 所以回到(0, 0)继续往下讨论。

步骤 5:





(1, 3)→放置成功

	0	1	2	3
0				
1				
2				
3				

步骤 6:

(2,0) → 放置失败(∵第 1 列已有 1 个皇后)

(2,1) → 放置成功

	0	1	2	3
0				
1				
2				
3				








步骤 7:

(3,0) → 放置失败(∵第 1 列已有 1 个皇后)

(3,1) → 放置失败(∵第 2 列已有 1 个皇后)

(3,2) → 放置失败(∵左上角已有 1 个皇后)

(3,3) → 放置失败(∵第 4 列已有 1 个皇后)

	0	1	2	3
0				
1				
2				
3				





步骤 8:

第 4 行无法放置任何皇后, 所以回到(1,3)继续往下讨论。

步骤 9:

(2,2) → 放置失败(∵右上方已有 1 个皇后)

(2,3) → 放置失败(∵第四列已有 1 个皇后)


	0	1	2	3
0				
1				
2				
3				

步骤 10:

第 3 行无法放置任何皇后, 所以回到起始位置继续往下讨论。

步骤 11:

(0,1) → 放置成功

	0	1	2	3
0				
1				
2				
3				






步骤 12:

(1,0) → 放置失败(∵右上角已有 1 个皇后)

(1,1) → 放置失败(∵第 2 列已有 1 个皇后)




(1,2) → 放置失败(∵左上角已有 1 个皇后)

(1,3) → 放置成功

	0	1	2	3
0				
1				
2				
3				

步骤 13:

(2,0) → 放置成功







	0	1	2	3
0				
1				
2				
3				

步骤 14:

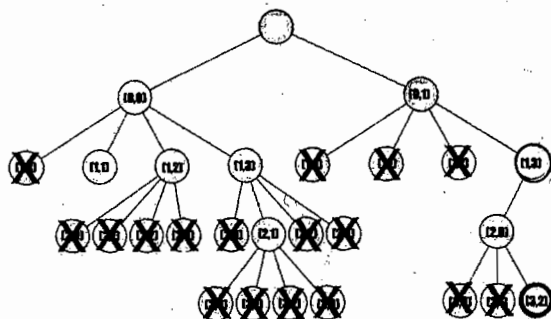
(3,0) → 放置失败(∵第1列已有1个皇后)

(3,1) → 放置失败(∵第2列已有1个皇后)

(3,2) → 放置成功

	0	1	2	3
0				
1				
2				
3				

由上列的实际操作后，我们可归纳出下列这张图：



程序实例:

运用递归来解 N 皇后问题。

程序构思:

必须先判断传入的坐标位置是否可放置皇后。(判断该坐标上、下、左、右、左上、右上、左下、右下八个方向是否有其它的皇后, 有返回 false, 无返回 true)。

假设传入的坐标为(LocX,LocY), 棋盘大小为 N*N。

坐标上方: (LocX,LocY-1)到(LocX,0)是否有其它的皇后。

坐标下方: (LocX,LocY+1)到(LocX,N-1)是否有其它的皇后。

坐标左方: (LocX-1,LocY)到(0,LocY)是否有其它的皇后。

坐标右方: (LocX+1,LocY)到(N-1,LocY)是否有其它的皇后。

坐标左上方: (LocX-1,LocY-1)到(LocX,0)或(0,LocY)是否有其它的皇后。

坐标右上方: (LocX+1,LocY-1)到(LocX,0)或(N-1,LocY)是否有其它的皇后。

坐标左下方: (LocX-1,LocY+1)到(LocX,N-1)或(0,LocY)是否有其它的皇后。

坐标右下方: (LocX+1,LocY+1)到(LocX,N-1)或(N-1,LocY)是否有其它的皇后。

递归结束条件:

当 N 个皇后皆放置成功。

递归执行部分:

判断传入坐标是否可放置皇后, 可以放置则依序递归放置下一个皇后。

程序源代码:

```

01      /* ===== Program Description ===== */
02      /* 程序名称: queen.c */
03      /* 程序目的: 运用递归来解 N 皇后问题 */
04      /* Written By Kuo-Yu Huang. (WANT Studio.) */
05      /* ===== */
06
07      char Chessboard[8][8]; /* 声明 8*8 的空白棋盘 */
08
09      /* -----*/
10      /* 递归解 N 皇后问题 */
11      /* -----*/
12      int N_Queens(int LocX,int LocY,int Queens)
13      {
14          int i,j; /* 循环计数变量 */
15          int Result=0;
16
17          if ( Queens == 8 ) /* 递归结束条件 */
18              return 1;
19          else /* 递归执行部分 */
20              if ( QueenPlace(LocX,LocY) )
21              {
22                  Chessboard[LocX][LocY] = 'Q';
23                  for (i=0;i<8;i++)

```

```

24         for (j=0;j<8;j++)
25         {
26             Result += N_Queens(i,j,Queens+1);
27             if (Result > 0)
28                 break;
29         }
30         if (Result > 0)
31             return 1;
32         else
33         {
34             Chessboard[LocX][LocY] = 'X';
35             Return 0;
36         }
37     }
38     else
39         return 0;
40 }
41
42 /* ----- */
43 /* 判断传入坐标是否可放置皇后 */
44 /* ----- */
45 int QueenPlace(int LocX,int LocY)
46 {
47     int i,j;
48
49     if (Chessboard[LocX][LocY] != 'X') /* 判断是否有皇后 */
50         return 0;
51
52     for (j=LocY-1;j>=0;j--) /* 判断上方是否有皇后 */
53         if (Chessboard[LocX][j] != 'X')
54             return 0;
55
56     for (j=LocY+1;j<8;j++) /* 判断下方是否有皇后 */
57         if (Chessboard[LocX][j] != 'X')
58             return 0;
59
60     for (i=LocX-1;i>=0;i--) /* 判断左方是否有皇后 */
61         if (Chessboard[i][LocY] != 'X')
62             return 0;
63
64     for (i=LocX+1;i<8;i++) /* 判断右方是否有皇后 */
65         if (Chessboard[i][LocY] != 'X')
66             return 0;
67
68     i = LocX - 1;
69     j = LocY - 1;
70     while ( i>=0 && j>=0 ) /* 判断左上方是否有皇后 */
71         if (Chessboard[i--][j--] != 'X')
72             return 0;
73
74     i = LocX + 1;
75     j = LocY - 1;
76     while ( i<8 && j>=0 ) /* 判断右上方是否有皇后 */
77         if (Chessboard[i++][j--] != 'X')
78             return 0;
79
80     i = LocX - 1;
81     j = LocY + 1;
82     while ( i>=0 && j<8 ) /* 判断左下方是否有皇后 */
83         if (Chessboard[i--][j++] != 'X')
84             return 0;

```

```

85
86     i = LocX + 1;
87     j = LocY + 1;
88     while ( i<8 && j<8 )          /* 判断右下方是否有皇后 */
89         if (Chessboard[i++][j++] != 'X')
90             return 0;
91     return 1;
92 }
93
94 /* ----- */
95 /* 主程序 */
96 /* ----- */
97 void main ()
98 {
99     int    i,j;          /* 循环计数变量 */
100
101     for (i=0;i<8;i++)
102         for (j=0;j<8;j++)
103             Chessboard[i][j] = 'X';
104     N_Queens(0,0,0);
105
106     printf("The graph of 8 Queens on the Chessboard.\n");
107     printf("      0  1  2  3  4  5  6  7 \n");
108     printf("  +---+---+---+---+---+---+---+---+\n");
109
110     for (i=0;i<8;i++)
111     {
112         printf("    %d |",i);
113         for (j=0;j<8;j++)
114             printf("-%c-",Chessboard[i][j]);
115         printf("\n  +---+---+---+---+---+---+---+---+\n");
116     }
117 }

```

运行结果:

```

C:\DS>queen
The graph of 8 Queens on the Chessboard.
      0  1  2  3  4  5  6  7
  +---+---+---+---+---+---+---+---+
0 | -Q-|-X-|-X-|-X-|-X-|-X-|-X-|-X-|
  +---+---+---+---+---+---+---+---+
1 | -X-|-X-|-X-|-X-|-Q-|-X-|-X-|-X-|
  +---+---+---+---+---+---+---+---+
2 | -X-|-X-|-X-|-X-|-X-|-X-|-X-|-Q-|
  +---+---+---+---+---+---+---+---+
3 | -X-|-X-|-X-|-X-|-X-|-Q-|-X-|-X-|
  +---+---+---+---+---+---+---+---+
4 | -X-|-X-|-Q-|-X-|-X-|-X-|-X-|-X-|
  +---+---+---+---+---+---+---+---+
5 | -X-|-X-|-X-|-X-|-X-|-X-|-Q-|-X-|
  +---+---+---+---+---+---+---+---+
6 | -X-|-Q-|-X-|-X-|-X-|-X-|-X-|-X-|
  +---+---+---+---+---+---+---+---+
7 | -X-|-X-|-X-|-Q-|-X-|-X-|-X-|-X-|
  +---+---+---+---+---+---+---+---+

C:\DS>

```


6.6 迷宫问题

迷宫问题是指在一个 $m \times n$ 的矩阵当中，其中“0”代表可以行走的区域、“1”代表不可行走的区域，当你处在迷宫的任何一个位置，除了不可走不可行走的区域外，其余皆可以往上、下、左、右、左上、右上、左下、右下八个方向行走来找寻迷宫出口。

假设有个 5×4 的迷宫如下：

入口	1	0	0	0
1	0	1	1	0
1	0	1	1	0
1	1	0	1	1
0	0	1	0	1
1	1	1	0	出口

面对这种复杂的问题，我们可从分析题目的过程当中归纳出其规则，看适不适合用运用递归程序来解，如何适合用递归程序来解，就必须归纳出递归结束条件和递归执行部分。

从这个迷宫问题中，我们发现不论玩家处于迷宫中任一角落皆可往目前位置周围的 8 个方向行走，我们对这个 8 个方向定义出行走的优先级及坐标位置如下：

左上 (X-1,Y-1) 8	上 (X-1,Y) 1	右上 (X-1,Y+1) 2
左 (X,Y-1) 7	目前位置 (X,Y)	右 (X+1,Y) 3
左下 (X+1,Y-1) 6	下 (X+1,Y) 5	右下 (X+1,Y+1) 4

由此，我们只要将每一个位置再区分为 8 个小步骤递归寻找出迷宫中可行的路，直到找一组解为止。假设现在玩家决定往右边行走，则右边的坐标便成为新的坐标，如此的递归遍历每一个可行的坐标。

我们现在先实际操作一个上述那个大小为 6×5 的迷宫，让读者熟悉一下迷宫的运作流程。一般的迷宫问题为了简化迷宫边界的判断，一个 6×5 的迷宫，常会声明成 8×7 大小的数组，并将边界地区设为不可行走的区域，如下图灰色部分：

	0	1	2	3	4	5	6
0							
1		0	1	0	0	0	
2		1	0	1	1	0	
3		1	0	1	1	0	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

如上图入口为(1,1), 出口为(6,5), 此时将(1,1)标记为已走过符号“2”, 走过但未能有通路的标记符号为“3”。

步骤 1:

由(1,1)往上、往右上、往右皆不可行走。

往右下可前进到(2,2), 将(2,2)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	0	0	0	
2		1	2	1	1	0	
3		1	0	1	1	0	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 2:

(2,2)往上不可行走。

往右上可前进到(1,3), 将(1,3)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	2	0	0	
2		1	2	1	1	0	
3		1	0	1	1	0	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 3:

(1,3)往上、往右上皆不可行走。

往右可前进到(1,4), 将(1,4)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	2	2	0	
2		1	2	1	1	0	
3		1	0	1	1	0	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 4:

(1,4)往上、往右上皆不可行走。

往右可前进到(1,5)，将(1,5)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	2	2	2	
2		1	2	1	1	0	
3		1	0	1	1	0	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 5:

(1,5)往上、往右上、往右、往右下皆不可行走。

往下可前进到(2,5)，将(2,5)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	2	2	2	
2		1	2	1	1	2	
3		1	0	1	1	0	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 6:

(2,5)往上、往右上、往右、往右下皆不可行走。

往下可前进到(3,5)，将(3,5)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	2	2	2	
2		1	2	1	1	2	
3		1	0	1	1	2	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 7:

(3,5)往上、往右上、往右、往右下、往下、往左下、往左、往左上皆不可行走，依次退回直到(2,2)，并将退回的坐标标记为“3”。

	0	1	2	3	4	5	6
0							
1		2	1	3	3	3	
2		1	2	1	1	3	
3		1	0	1	1	3	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 8:

(2,2)往上、往右上、往右、往右下皆不可行走。

往下可前进到(3,2)，将(3,2)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	3	3	3	
2		1	2	1	1	3	
3		1	2	1	1	3	
4		1	1	0	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 9:

(3,2)往上、往右上、往右皆不可行走。

往右下可前进到(4,3)，将(4,3)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	3	3	3	
2		1	2	1	1	3	
3		1	2	1	1	3	
4		1	1	2	1	1	
5		0	0	1	0	1	
6		1	1	1	0	0	
7							

步骤 10:

(4,3)往上、往右上、往右皆不可行走。

往右下可前进到(5,4)，将(5,4)标记为已走过符号“2”。

	0	1	2	3	4	5	6
0							
1		2	1	3	3	3	
2		1	2	1	1	3	
3		1	2	1	1	3	
4		1	1	2	1	1	
5		0	0	1	2	1	
6		1	1	1	0	0	
7							

步骤 11:

(5,4)往上、往右上、往右皆不可行走。

往右下可前进到(6,5)，将(6,5)标记为已走过符号“2”。

此时已达出口(6,5)。

	0	1	2	3	4	5	6
0							
1		2	1	3	3	3	
2		1	2	1	1	3	
3		1	2	1	1	3	
4		1	1	2	1	1	
5		0	0	1	2	1	
6		1	1	1	0	0	
7							

程序实例:

运用递归来解迷宫问题。

程序构思:

递归结束条件:

当已经走到出口时。(当出口(6,5), 标记为 2 时)

递归执行部分:

判断传入坐标是否可走。

如果可以走的话, 则递归调用往上、往右上、往右、往右下、往下、往左下、往左、往左上坐标是否可走。

可走的话, 返回 1。不可走的话, 标记改为 3(已走过, 但未能有通路)。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: maze.c */
03  /* 程序目的: 运用递归来解迷宫问题 */
04  /* Written By Kuo-Yu Huang. (WANT Studio.) */
05  /* ===== */
06
07  int Maze[8][7] = { /* 声明 5*4 的迷宫, 外围不可走 */
08      1, 1, 1, 1, 1, 1, 1,
09      1, 0, 1, 0, 0, 0, 1,
10      1, 1, 0, 1, 1, 0, 1,
11      1, 1, 0, 1, 1, 0, 1,
12      1, 1, 1, 0, 1, 1, 1,
13      1, 0, 0, 1, 0, 1, 1,
14      1, 1, 1, 1, 0, 0, 1,
15      1, 1, 1, 1, 1, 1, 1};
16
17  /* ----- */
18  /* 递归解迷宫问题 */
19  /* ----- */
20  int Way(int LocX, int LocY)
21  {
22      if ( Maze[6][5] == 2 ) /* 递归结束条件 */
23          return 1;
24      else /* 递归执行部分 */
25          if ( Maze[LocY][LocX] == 0 )
26          {
27              Maze[LocY][LocX] = 2;
28              if ( Way(LocX, LocY-1) )
29                  return 1;
30              else if ( Way(LocX+1, LocY-1) )
31                  return 1;
32              else if ( Way(LocX+1, LocY) )
33                  return 1;
34              else if ( Way(LocX+1, LocY+1) )
35                  return 1;
36              else if ( Way(LocX, LocY+1) )
37                  return 1;
38              else if ( Way(LocX-1, LocY+1) )
39                  return 1;

```

```

40         else if ( Way(LocX-1,LocY) )
41             return 1;
42         else if ( Way(LocX-1,LocY-1) )
43             return 1;
44         else
45         {
46             Maze[LocY][LocX] = 3;
47             return 0;
48         }
49     }
50     else
51         return 0;
52 }
53
54 /* ----- */
55 /* 主程序 */
56 /* ----- */
57 void main ()
58 {
59     int i,j;          /* 循环计数变量 */
60
61     printf("==Problem of Maze ==\n");
62     printf("The Maze source is (1,1).\n");
63     printf("The Maze Destination is (6,5).\n");
64     Way(1,1);
65
66     printf("The graph of Maze.\n");
67     printf("    0  1  2  3  4  5  6  \n");
68     printf("  +---+---+---+---+---+---+\n");
69
70     for (i=0;i<8;i++)
71     {
72         printf("  %d |",i);
73         for (j=0;j<7;j++)
74             printf("-%d-",Maze[i][j]);
75         printf("\n  +---+---+---+---+---+---+\n");
76     }
77 }

```

运行结果:

```

C:\DS>maze
==Problem of Maze ==
The Maze source is (1,1).
The Maze Destination is (6,5).
The graph of Maze.
    0  1  2  3  4  5  6
  +---+---+---+---+---+---+
0 |-1-|-1-|-1-|-1-|-1-|-1-|-1-|
  +---+---+---+---+---+---+
1 |-1-|-2-|-1-|-3-|-3-|-3-|-1-|
  +---+---+---+---+---+---+
2 |-1-|-1-|-2-|-1-|-1-|-3-|-1-|
  +---+---+---+---+---+---+
3 |-1-|-1-|-2-|-1-|-1-|-3-|-1-|
  +---+---+---+---+---+---+
4 |-1-|-1-|-1-|-2-|-1-|-1-|-1-|
  +---+---+---+---+---+---+
5 |-1-|-0-|-0-|-1-|-2-|-1-|-1-|
  +---+---+---+---+---+---+

```

```

6  |-1-|-1-|-1-|-1-|-1-|-0-|-2-|-1-|
   +---+---+---+---+---+---+---+
7  |-1-|-1-|-1-|-1-|-1-|-1-|-1-|-1-|
   +---+---+---+---+---+---+---+

```

C:\DS>

【习题】

一、复习

1. 递归程序设计时，必须要有递归结束条件，否则会陷入死循环中。
2. 递归程序利用堆栈来记录函数调用后的返回地址。
3. 递归程序的时间复杂度和空间复杂度比非递归程序有效率。
4. 有一个递归程序如下：

```

int fact(int N)
{
    if (n<=0)
        return 1;
    else
        return N * fact(N-1);
}

```

下列语句何者正确？

- (a) 这个递归函数，若计算 $\text{fact}(n)$ ，则需要执行 n 次。
 - (b) $\text{fact}(7) = 5040$ 。
 - (c) 这个递归程序最多只能计算到 $\text{fact}(8)$ 。
 - (d) 以上都不是。
5. 有一个递归程序如下：

```

int X(int N)
{
    if (N <= 3)
        return 1;
    else
        return X(N-2) + X(N-4) + 1;
}

```

请问 $X(X(8))$ 需要执行几次 X 函数？

- (a) 8 次
 - (b) 9 次
 - (c) 16 次
 - (d) 18 次
6. 汉诺塔问题，若 A 塔有 3 个铁盘，欲全搬至 C 塔，则下列语句哪一句错误？(复选)
 - (a) 第 1 步为将 1 号铁盘从 A 塔搬至 B 塔。
 - (b) 第 4 步为将 3 号铁盘从 A 塔搬至 C 塔。
 - (c) 第 7 步为将 1 号铁盘从 A 塔搬至 C 塔。
 - (d) 需要 8 次才能完成工作。
 7. 有一个递归程序如下：

```

int maze(int a,int b,int c)
{
    if (a<b)
        return a;
}

```



```

else
    return c*maze(a/b,b,c) + (a % b);
}

```

下列语句何者正确。

- (a) $\text{maze}(1020,10,7) = 356$
- (b) $\text{maze}(352,4,11) = 16214$
- (c) $\text{maze}(16,2,2) = 8$
- (d) 以上都不是。

二、应用

1. 试举例说明堆栈在递归程序上的用途。
2. 试说明当一个递归程序，缺少递归结束条件时可能发生的情形。
3. 假设有一数学函数如下：

$A(m,n) = n+1,$	$m=0$
$A(m,n) = A(m-1,1),$	$m \neq 0 \text{ and } n=0$
$A(m,n) = A(m-1,A(m,n-1)),$	otherwise

试设计一个递归程序来解 $A(3,3)$ 和 $A(5,2)$ 。

4. 假设有 n 个元素，试利用递归程序求出全部的排列方式。
5. 试设计出一个递归程序，求出 n 笔数据中，第 $n/2$ 大的数据。
6. 假设有一数学函数：当 $n=3k$ ，且 $k \geq 1$ 时， $f(n)=2f(n/3)+4$ ，已知 $f(1)=2$ ，求 $f(729)$ 。

基础树状结构

第 7 章

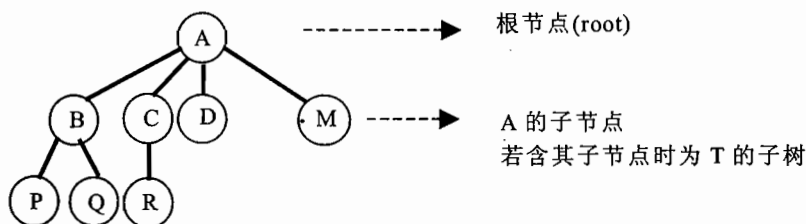
- ◆ 何谓树状结构
- ◆ 二叉树
- ◆ 二叉树表示法
- ◆ 二叉树的遍历
- ◆ 二叉树的建立(递归法)
- ◆ 二叉树的查找
- ◆ 二叉树的节点删除
- ◆ 二叉树的复制
- ◆ 二叉树的比较
- ◆ 二叉树的映像
- ◆ 一般树转二叉树
- ◆ 引线二叉树
- ◆ 二叉树的应用(表达式)

7.1 何谓树状结构

7.1.1 何谓树

树状结构是由一个或多个节点所构成之有限集合。每一棵树必有一特定的节点，称做根节点(root)。根节点之下可以有零个以上的子节点(可以没有)，而各子节点也可为子树，拥有自己的子节点。

如图所示：树 T

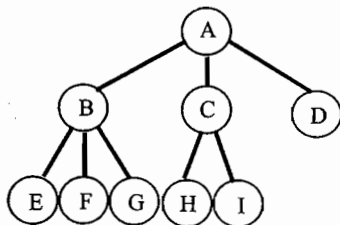


从图中树 T 可知，节点 A 为树 T 的根节点(root)，B,C,D,...,M 则为节点 A 的子节点，若包含其下拥有的所有子节点，则为 Tree—T 的子树(subtree)。例如 B 是 A 的子节点，P、Q 皆是 B 的子节点，而 B、P、Q 为树 T 的子树。

若一棵树中的节点最多可以有 n 个子节点，则称这样的树为 n 元树。例如二叉树中的节点，最多只能有两个子节点。

7.1.2 树的相关名称及意义

- (1) 根节点 (root node):
一棵树中没有父节点的节点，称为根节点。
- (2) 叶节点 (leaf node)或终端节点 (terminal mode):
一棵树中没有子节点的节点，称为叶节点。
- (3) 非终端节点 (nonterminal mode)
除了叶节点以外的其它节点，称为非终端节点。



根节点：A
叶节点：D、E、F、G、H、I
非终端节点：B、C

- (4) 父节点 (parent)和子节点 (child):
若节点 x 有一个以节点 y 为树根(root)的子树，则 x 为 y 的父节点，而 y 为 x 的子节点。
- (5) 兄弟 (sibling):

同一个父节点之节点，称为兄弟。

如图，B、C、D的父节点均为A，故B、C、D互为兄弟。

(6) 分支度 (degree):

每个节点所拥有的子节点个数。而一棵树中最大的分支度值，即为该树的分支度。

如图，A的分支度为3，B的分支度为3，C的分支度为1，其中最大的分支度值为3，故树T的分支度为3。

(7) 阶层 (level):

阶层为节点之特性值，将根节点之阶层设为1，其子节点为2，依此类推。

如图，阶层为1有节点A，阶层为2有节点B、C、D，阶层为3有节点E、F、G、H、I。

(8) 高度 (height)或深度 (depth):

一棵树中的最大阶层值，称为树的高度或深度。

如图，最大阶层值为3，故树T之高度为3。

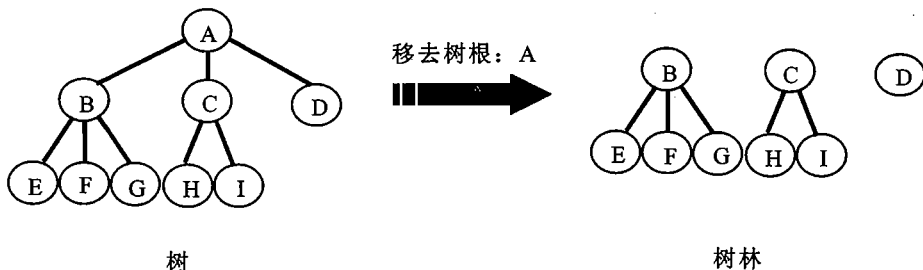
(9) 祖先 (ancestor):

由某节点x到根节点之路径上的所有节点，均称为x之祖先。

如图，节点I到根节点路径上节点A、C即为I的祖先。

(10) 树林 (forest):

$n \geq 0$ 个树的集合称为树林。若将一树的根节点移去，所剩这恰是一树林。



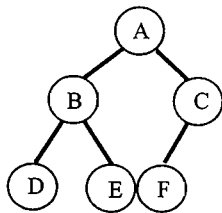
7.2 二叉树

7.2.1 何谓二叉树

二叉树(Binary tree)是树的一种，二叉树中的节点至多只能有两个子节点。

二叉树的定义如下：

- (1) 由有限个节点所构成之集合，此集合可以为空的。
- (2) 二叉树的根节点下可分成两个子树，称为左子树和右子树，左子树和右子树亦称二叉树。



如图, 该二叉树以 A 为根节点, 其左子树和右子树分别为:



由于二叉树之子树有顺序关系, 分为左子树和右子树, 所以下面两棵树是相同的树, 但却是不同的二叉树。



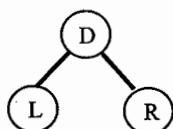
7.2.2 二叉树和树的比较

- (1) 二叉树可为空, 而树不可以(至少要有根节点)
- (2) 二叉树的子树有顺序关系, 而树没有
- (3) 二叉树的分支度必为 0、1 或 2, 而树的分支度可大于 2

7.2.3 二叉树的相关特色

- (1) 歪斜树 (skewed binary tree)

一般二叉树可视为:



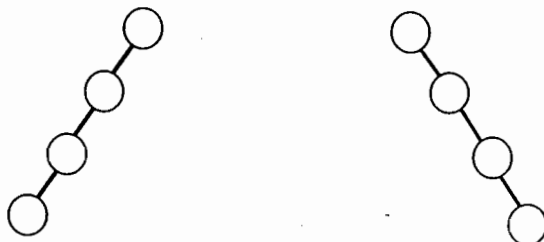
D: 根节点

L: 左子树

R: 右子树

其所有子树的结构也是如此。在一棵树中, 若所有节点的 L 均不存在, 则此树为右歪斜树(right skewed binary tree), 反之, 所有节点的 R 均不存在, 则此树为左歪斜树(left skewed binary tree)。

例如:



左歪斜树

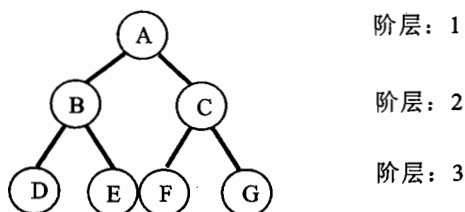
右歪斜树

- (2) 满二叉树 (full binary tree)

一树中所有叶节点均在同一阶层, 而其它非终端节点(nonterminal node)之分支度均为 2, 则此

树为一满二叉树。若该树的高度为 h ，则此二叉树的节点为 $2^h - 1$ 。

例如：

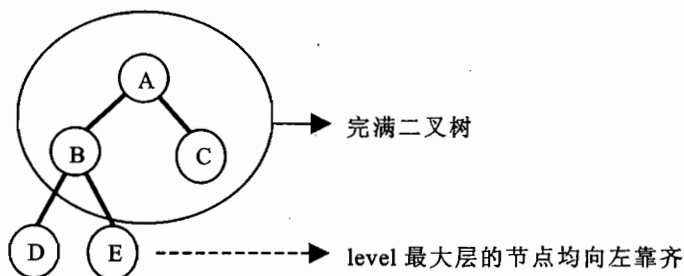


图中二叉树之高度为 3，节点数为 $2^3 - 1 = 7$ ，为满二叉树。

(3) 完全二叉树 (complete binary tree)

一棵树扣除掉最大阶层那层后为一满二叉树，且阶层最大那层的节点均向左靠齐，则该二叉树称为完全二叉树。

例如：



图中若扣除 level 3 后，A、B、C 为一满二叉树，且 D、E 在 level 3 中均向左靠齐，故其为一完全二叉树。

- (4) 阶层(level)为 i 的二叉树，最多有 2^{i-1} 个节点。
- (5) 高度(height)为 h 的二叉树，最多有 $2^h - 1$ 个节点。
- (6) 对任一个非空的二叉树而言，若分支度为 i 的节点各有 n_i 个，则

$$n_0 = n_2 + 1$$

证明： $n = n_0 + n_1 + n_2$

$$\text{Branch} = n - 1 = n_0 + n_1 + n_2 - 1 \quad \text{-----(1)}$$

$$\text{Branch} = 0 \cdot n_0 + 1 \cdot n_1 + 2 \cdot n_2 \quad \text{-----(2)}$$

$$= n_1 + 2 \cdot n_2$$

由(1)(2)可得：

$$n_0 + n_1 + n_2 - 1 = n_1 + 2 \cdot n_2$$

$$\rightarrow n_0 = n_2 + 1$$

7.3 二叉树表示法

二叉树节点的表示法，常用的有下列 3 种方法：

- (1) 二叉树数组表示法
- (2) 二叉树结构数组表示法
- (3) 二叉树链表表示法

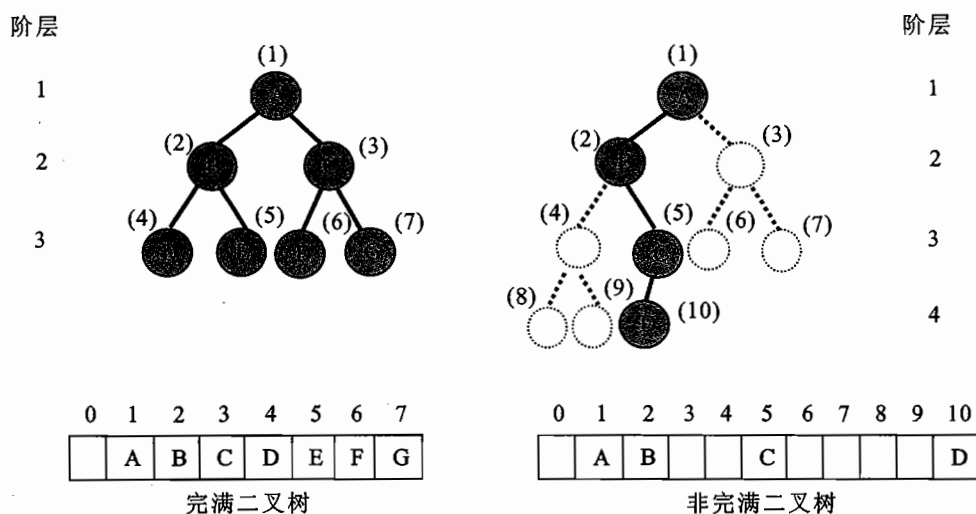
其中“数组表示法”和“结构数组表示法”是属于静态内存空间配置，而“链表表示法”是利用列表结构的方式，属于动态内存空间配置。

下面小节将对这3种二叉树表示法做更详细的说明。

7.3.1 二叉树数组表示法

若一个二叉树的树高 h ，则根据前面所介绍过的，当其为满二叉树时会拥有最多节点 2^h-1 个。在满二叉树中，可对各阶层的节点由低阶层到高层阶，由左到右，从1开始依序编号，再根据编号存入相对应索引编号之数组中。若该树不为满二叉树，也可对各节点编成在满二叉树中相同位置之节点编号值，再以相同的方式存入数组中；若某一编号没有节点存在，则不存值于数组中。

例如：



从二叉树数组表示法的节点编号来看，假设一父节点编号为 n ，可以得到：

- (1) 左子节点为父节点乘以 2: $2*n$
- (2) 右子节点为父节点乘以 2 加 1: $2*n+1$

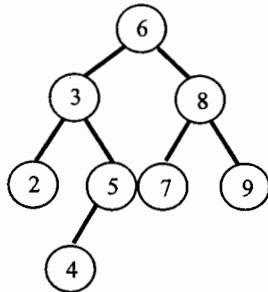
例如左图中 C 之节点编号 3，为 F、G 的父节点，左子节点 F 编号为 $2*3=6$ ，而右子节点 G 之编号为 $2*3+1=7$ 。故若要用 1 个一维数组来代表 1 棵二叉树，所需的数组长度为 $2^{\text{Max}}-1$ ，其中 Max 为最大可能的阶层。

建立二叉树节点数据的原则如下：

- (1) 以第 1 个建立之元素为根节点
- (2) 依序将元素值与根节点做比较
 - (a) 若元素值大于根节点值，则将元素值往根节点之右子节点移动，若此右子节点为空，则将元素值存入；否则就重复比较，直到找到适当之空节点为止。
 - (b) 若元素值小于根节点值，则将元素值往根节点之左子节点移动，若此左子节点为空，则将元素值存入；否则就重复比较，直到找到适当之空节点为止。

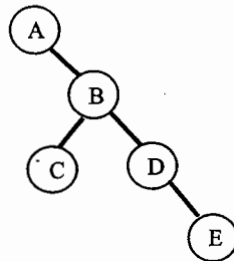
事实上，上述的二叉树建立原则即满足了二叉查找树(Binary search tree)的条件，关于二叉查找树的内容将在 7.7 节中再做更详细的介绍。

例如：输入数据的顺序为 6、3、8、5、2、9、4、7 欲建立一个二叉树，得到结果如下图：



二叉树数组表示法的优点是：对于任一个节点都能很容易地找到其父节点、子节点及兄弟，而且每个节点的存储空间不大，只占用数组的一个内存空间。但当二叉树之深度和节点数之比例偏高时(二叉树分布不均匀，如歪斜树)，则内存的利用率会偏低，容易造成空间的浪费。

例如：



在数组中的分布如下：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A		B			C	D								E

由于节点分布不均匀，造成内存的使用率偏低。另外，由于数组表示法是用循序的方式处理，故在插入或删除节点时，需要移动其它元素方可完成。

程序实例：

依序输入元素值，以数组方式建立二叉树，并输出节点内容。

程序构思：

先依序输入元素值，并存入数组 `nodelist` 中，再一一建立成二叉树数组 `b_tree`，其中根节点为索引值 1，其余节点的建立则遵守左小($level*2$)右大($level*2+1$)的原则，最后输出所建立二叉树之节点内容。



程序源代码：

```
01  /* ===== Program Description ===== */
02  /* 程序名称: Tree_01.c */
03  /* 程序目的: 二叉树的数组表示法 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
```



```
05  /* ===== */
06  /*-----*/
07  /*      建立二叉树      */
08  /*-----*/
09  void create_btree(int b_tree, int *nodelist, int len)
10  {
11      int i;
12      int level;    /*树的阶层数*/
13
14      b_tree[1] = nodelist[1];    /*以第一个元素为根节点*/
15      for (i=2; i<=len; i++)    /*依序建立其它节点*/
16      {
17          level=1;    /*从阶层 1 开始建立*/
18          while ( b_tree[level] != 0)    /*判断是否有子树存在*/
19          {
20              if (nodelist[i] < b_tree[level])    /*判断是左子树还是右子树*/
21                  level = level * 2;    /*左子树*/
22              else
23                  level = level * 2 + 1;    /*右子树*/
24          }
25          b_tree[level] = nodelist[i];    /*将元素值存入节点*/
26      }
27  }
28  /*-----*/
29  /*主程序:输入元素、建立数组二叉树并输出二叉树内容*/
30  /*-----*/
31  void main ( )
32  {
33      int i, index;
34      int data;    /*读入输入值所使用的暂存变量*/
35      int b_tree[16];    /*声明二叉树数组*/
36      int nodelist[16];    /*声明存储输入数据之数组*/
37
38      printf("\n Please input the elements of binary tree(Exit for
39      0):\n");
40      index=1;
41
42      /*-----读取数值存到数组中-----*/
43      scanf("%d", &data);    /*读取输入值存到变量 data*/
44
45      while (data != 0)    /*读取尚未结束*/
46      {
47          nodelist[index]=data;
48          index=index+1;
49          scanf("%d",&data);
50      }
51
52      /*-----清除二叉树数组的内容-----*/
53      for (i=1; i<16; i++)
54          b_tree[i]=0;
55
56      /*-----建立二叉树-----*/
57      create_btree(b_tree, nodelist, index);
58
59      /*-----输出二叉树的内容-----*/
60      printf("\n\nThe binary tree is:\n");
61      for (i=1; i<16; i++)
62          printf("%2d: [%d] \n", i, b_tree[i]);
63  }
```

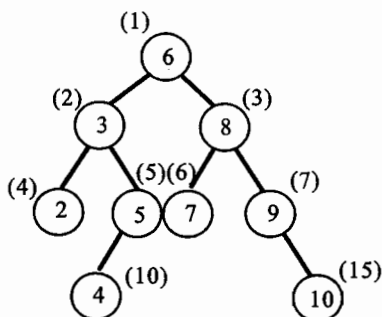
运行结果:

```
C:\DS>Tree_01
Please input the elements of binary tree(Exit for 0):
6 3 8 5 2 9 4 7 10 0

The binary tree is:
1: [6]
2: [3]
3: [8]
4: [2]
5: [5]
6: [7]
7: [9]
8: [0]
9: [0]
10 : [4]
11 : [0]
12 : [0]
13 : [0]
14 : [0]
15 : [10]

C:\DS>
```

运行结果相对应的树状结构如下:



7.3.2 二叉树结构数组表示法

二叉树中的节点最多只能有两个子节点,我们可用结构的类型来声明节点的存储方式。此结构包含 3 个字段,其中一个字段是用来存放节点的数据内容,而另两个字段则是分别存放左子树和右子树在数组中的索引值。

结构数组的类型如图:

Left	Data	Right
------	------	-------

data: 存放节点的数据内容

left: 存放左子树在数组中的索引值

right: 存放右子树在数组中的索引值

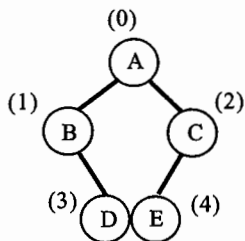
二叉树的结构声明如下:

```

struct tree
{
    int left;
    char data;
    int right;
}
typedef struct tree treenode;
treenode b_tree[15];

```

声明完成后, `b_tree` 即是用来存放二叉树各节点之结构数组。在结构数组中, 会将根节点置于数组结构中索引值为 0 之处, 将节点值存在 `data` 字段, 而 `left` 及 `right` 字段则分别存储左右子树在数组结构中的索引值, 若子树不存在则存值-1。例如, 有一棵二叉树其树状结构与结构数组表示法如下:



索引值	left	date	Right
0	1	A	2
1	-1	B	3
2	4	C	-1
3	-1	D	-1
4	-1	E	-1

图中根节点为 A, 故 A 在结构数组索引值为之处, 其左子节点 B 在索引值 1, 右子节点在索引值 2, 故节点 A 的 `left` 字段和 `right` 字段分别是 1 和 2。而节点 C 的 `right` 字段值为-1, 表示其没有右子节点。而节点 D 和 E 都是叶节点(leaf node)没有子节点, 故 `left` 和 `right` 字段均为-1。

程序实例:

依序输入元素值, 以结构数组方式建立二叉树, 并输出节点内容。

程序构思:

先依序输入元素值, 并存入数组 `nodelist` 中, 再一一建立成二叉树数组 `b_tree`, 其中根节点为索引值 1, 其余节点的建立则遵守左字段存左子节点之索引值, 右字段存右子节点的原则, 最后输出所建立二叉树之节点内容。

程序源代码:

```

01      /* ===== Program Description ===== */
02      /* 程序名称: Tree_02.c */
03      /* 程序目的: 二叉树的结构数组表示法 */
04      /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05      /* ===== */
06      #include <stdlib.h>
07
08      struct tree
09      {
10          int left;
11          int data;
12          int right;
13      };
14      typedef struct tree treenode;
15      treenode b_tree[15];
16
17      /*-----*/
18      /*          建立二叉树          */
19      /*-----*/

```

```

19 void create_btree(int *b_tree, int *nodelist, int len)
20 {
21     int i;
22     int level;           /*树的阶层数*/
23     int position;        /*左树-1,右树1*/
24
25     b_tree[0].data=nodelist[0]; /*以第一个元素为根节点*/
26     for ( I=1;i<len; i++)      /*依序建立其它节点*/
27     {
28         b_tree[i].data=nodelist[i]; /*将元素值存入节点*/
29         level=0;                  /*从树根开始建立*/
30         position=0;               /*设置 position 值*/
31         while (position ==0)      /*寻找节点位置*/
32         {
33             /*判断是左子树或是右子树*/
34             if (nodelist[i] > b_tree[level].data)
35                 /*右树是否有下一阶层*/
36                 if (b_tree[level].right != -1)
37                     level=b_tree[level].right;
38                 else
39                     position=-1; /* 设置为右树*/
40             else
41                 /*左树是否有下一阶层*/
42                 if (b_tree[level].left != -1)
43                     level=b_tree[level].left;
44                 else
45                     position=1; /* 设置为左树*/
46         }
47         if (position == 1)        /*建立节点的左右连结*/
48             b_tree[level].left=i; /*连结左子树*/
49         else
50             b_tree[level].right=i; /*连结右子树*/
51     }
52 }
53 /*-----*/
54 /*主程序:输入元素、建立结构数组二叉树并输出二叉树内容 */
55 /*-----*/
56 void main ( )
57 {
58     int i,index;
59     int data;           /*读入输入值所使用的暂存变量*/
60     int nodelist[16];   /*声明存储输入数据之数组*/
61
62     printf("\n Please input the elements of binary tree(Exit for 0):\n");
63     index=1;
64
65     /*-----读取数值存到数组中-----*/
66     scanf("%d", &data); /*读取输入值存到变量 data*/
67
68     while (data != 0 )    /*读取尚未结束*/
69     {
70         nodelist[index]=data;
71         index=index+1;
72         scanf("%d",&data);
73     }
74
75     /*-----清除结构数组的内容-----*/
76     for (i=0; i<15; i++)
77     {
78         b_tree[i].data=0;
79         b_tree[i].left=-1;
80         b_tree[i].right=-1;

```

```

80     }
81
82     /*-----建立二叉树-----*/
83     create_btree(nodelist,index);
84
85     /*-----输出二叉树的内容-----*/
86     printf("\nThe binary tree content:\n");
87     printf("    left    data    right    \n");
88     printf("===== \n");
89     for (i=1; i<15; i++)
90         printf("%2d: [%2d] [%2d] [%2d] \n", i,
91             b_tree[i].left,b_tree[i].data, b_tree[i].right);
92     }

```

运行结果:

```

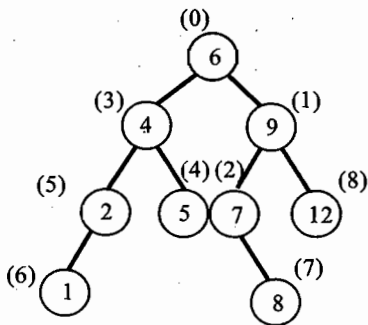
C:\DS>Tree_02
Please input the elements of binary tree(Exit for 0):
6 4 9 2 5 7 12 1 8 0

The binary tree content:
left data right
=====
0:  [ 2] [ 6] [ 1]
1:  [-1] [ 9] [ 8]
2:  [ 5] [ 7] [ 7]
3:  [-1] [ 4] [ 4]
4:  [-1] [ 5] [-1]
5:  [ 6] [ 2] [-1]
6:  [-1] [ 1] [-1]
7:  [-1] [ 8] [-1]
8:  [-1] [12] [-1]

C:\DS>

```

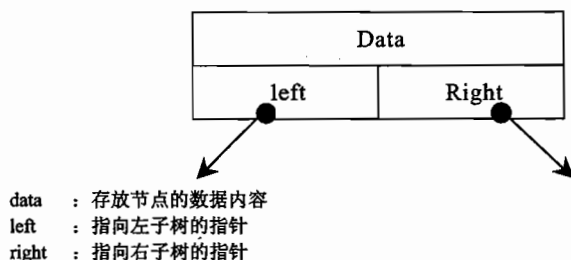
运行结果相对应的树状结构如下:



图中节点旁的数字为该节点在结构数组中的索引值, 没有节点的值都是-1, 表示没有子节点。这样的结构可以改善二叉树数组表示法中, 若要插入或删除节点需要移动大量数据的问题。因为有两个字段 left 和 right 来存储左右子树的索引值, 所以插入或删除节点时只要改变这两个字段值, 而不需要大量移动数据。

7.3.3 二叉树链表表示法

二叉树的“链表表示法”和“结构数组表示法”很相似，都是用 3 个字段来存储节点信息。两者之不同在于结构数组是以循序静态的方式处理，而链表是运用动态内存配置的方法来建立二叉树，其中左右字段是用来连结左右子树之指针。链表的节点结构如下：



二叉树链表结构的声明如下：

```
struct tree
{ struct tree *left;
  int data;
  struct tree *right;
}
typedef struct tree treenode;
treenode *b_tree;
```

程序实例：

依序输入元素值，以链表方式建立二叉树，并输出节点内容。

程序构思：

先依序输入元素值，并存入数组 `nodelist` 中，再一一建立成二叉树链接 `b_tree`，节点的建立是将左指针指向左子节，右指针指向右子节点，最后输出所建立二叉树的节点内容。

程序源代码：

```
01  /* ===== Program Description ===== */
02  /* 程序名称: Tree_03.c */
03  /* 程序目的: 二叉树的链表表示法 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07
08  struct tree /*声明树的结构*/
09  { struct tree *left; /*存放左子树的指针*/
10    int data; /*存放节点数据内容*/
11    struct *right; /*存放右子树的指针*/
12  };
13  typedef struct tree treenode; /*声明新类型树结构*/
14  treenode *b_tree; /*声明二叉树的链表*/
15
16  /*-----*/
17  /* 插入二叉树的节点 */
18  /*-----*/
```

```

18  /*-----*/
19  b_tree insert_node(b_tree root,int node)
20  {
21      b_tree newnode;          /*声明树根指针*/
22      b_tree currentnode;      /*声明目前节点指针*/
23      b_tree parentnode;       /*声明父节点指针*/
24
25      /*建立新节点的内存空间*/
26      newnode=(b_tree) malloc (sizeof(treenode));
27
28      newnode->data =node;      /*存入节点内容*/
29      newnode->right=NULL;      /*设置右指针初值*/
30      newnode->left=NULL;       /*设置左指针初值*/
31
32      if (root == NULL)
33          return newnode;      /*返回新节点的位置*/
34      else
35      {
36          currentnode=root;     /*存储目前节点指针*/
37          while ( currentnode !=NULL)
38          {
39              parentnode=currentnode; /*存储父节点指针*/
40              if (currentnode->data > node) /*比较节点的数值大小*/
41                  currentnode=currentnode->left; /*左子树*/
42              else
43                  currentnode=currentnode->right; /*右子树*/
44          }
45          if (parentnode->data > node) /*将父节点和子节点做连结*/
46              parentnode->left = newnode; /*子节点为父节点之左子树*/
47          else
48              parentnode->right= newnode; /*子节点为父节点之右子树*/
49      }
50      return root;              /*返回根节点之指针*/
51  }
52
53  /*-----*/
54  /*          建立二叉树          */
55  /*-----*/
56  b_tree create_btrees(int *data,int len)
57  {
58      b_tree root=NULL;         /*根节点指针*/
59      int i;
60
61      for (i=0; i <len; i++)     /*建立树状结构*/
62          root=insert_node(root,data[i]);
63      return root;
64  }
65
66  /*-----*/
67  /*          打印二叉树          */
68  /*-----*/
69  void print_btrees(b_tree root)
70  {
71      b_tree pointer;
72
73      pointer = root->left;
74      printf("Print left subtree node of root:\n");
75      while (pointer!=NULL)
76      {
77          printf("[%2d]\n",pointer->data); /*打印节点的内容*/
78          pointer=pointer->left;           /*指向左子节点*/

```

```

79     }
80
81     pointer = root->right;
82     printf("Print right_subtree node of root:\n");
83     while (pointer!=NULL)
84     {
85         printf("[%2d]\n",pointer->data);    /*打印节点的内容*/
86         point=point->right;                /*指向左子节点*/
87     }
88 }
89
90 /*-----*/
91 /*主程序:输入元素、建立链表二叉树并输出二叉树内容*/
92 /*-----*/
93 void main( )
94 {
95     b_tree root=NULL;                    /*树根指针*/
96
97     int i,index;
98     int value;                            /*读入输入值所使用的暂存变量*/
99     int nodelist[20];                    /*声明存储输入数据之数组*/
100
101     printf("\n Please input the elements of binary tree(Exit for
102     0):\n");
103     index=0;
104     /*-----读取数值存到数组中-----*/
105     scanf("%d", &value);                /*读取输入值存到变量 value*/
106
107     while (data != 0 )                    /*读取尚未结束*/
108     {   nodelist[index]= value;
109         index=index+1;
110         scanf("%d",&value);
111     }
112
113     /*-----建立二叉树-----*/
114     root=create_btree(nodelist,index);
115
116     /*-----打印二叉树节点内容-----*/
117     print_btree(root);
118 }

```

运行结果:

```

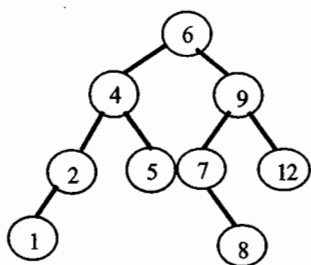
C:\DS>Tree_03
Please input the elements of binary tree(Exit for 0):
6 9 7 4 5 2 1 8 12 0

Print left_subtree node of root:
[4]
[2]
[1]
Print left_subtree node of root:
[9]
[12]

C:\DS>

```

运行结果相对应的树状结构如下:



由于进行打印时只印从根节点一直往左到 NULL 所经过的点 4、2、1，及从根节点一直往右到 NULL 所经过的点 9、12，不是整棵树的所有节点。关于二叉树的节点打印将在 7.4 节介绍前序、中序及后序的方法。

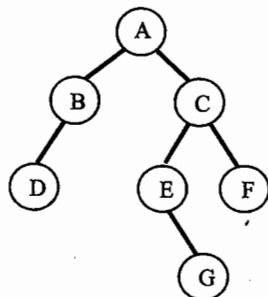
7.4 二叉树的遍历

“遍历”是抽取数据结构中的各个数据值，例如：数组和链表可从前端到尾端或从尾端至前端依序抽取各个数据值。而二叉树是一种特殊的数据结构，每个节点其下又各有左、右两个分支。“二叉树的遍历”是以固定的顺序，有系统地抽取二叉树中的各节点，且每个节点均恰好被抽取一次。

如上图所示，每个节点均有左右两个分支，在遍历的过程中可以选择往左或往右走，遍历结束，每个节点恰被抽取一次。事实上，二叉树的遍历是以递归的方式进行，依递归的调用顺序之不同，可分为下列 3 种不同的遍历方式：

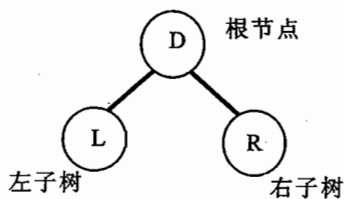
1. 前序遍历方式
2. 中序遍历方式
3. 后序遍历方式

在下面小节中，将分别对 3 种遍历方式做更详细的说明。

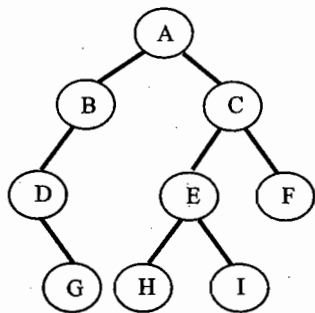


7.4.1 二叉树的前序遍历

前序遍历(Preorder traversal)是先遍历根节点，再遍历左子树，最后才遍历右子树。若一棵二叉树如左，则前序遍历的顺序为：DLR，也就是说每当遍历一个节点就先处理该节点，之后先向左方前进，直到无法前进才往右方走。



右图中从根节点 A 先往左子树 B 再到 D, 由于 D 没有左子树, 故转向右子树 G。再回到 B, 因为 B 没有右子树, 所以此时 A 之左子树均遍历完毕, 则转向 A 之右子树 C, 再往左边继续遍历, 依此类推, 故可得到前序遍历的顺利为: ABDGCEHIF。



前序遍历的步骤如下:

```

if 指向根节点的指针=NULL then
    此为空树, 遍历结束
else
    (1) 处理目前的节点
    (2) 往左走, 递归处理 preorder(root→left)
    (3) 往右走, 递归处理 preorder(root→right)
  
```

程序实例:

建立二叉树, 并以前序遍历的方式将节点内容输出。

程序构思:

输入元素值后建立二叉树, 以递归的方式做前序遍历, 其顺序为: 节点→左子树→右子树, 并将二叉树节点内容输出。

程序源代码:

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Tree_04.c */
03  /* 程序目的: 二叉树前序遍历方式 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07
08  struct tree          /*声明树的结构*/
09  { struct tree *left;  /*存放左子树的指针*/
10    int data;          /*存放节点数据内容*/
11    struct tree *right; /*存放右子树的指针*/
12  };
13  typedef struct tree treenode; /*声明新类型树结构*/
14  typedef treenode *b_tree;     /*声明二叉树的链表*/
15
16  /*-----*/
17  /*      插入二叉树的节点      */
18  /*-----*/
19  b_tree insert_node(b_tree root,int node)
20  {
21      b_tree newnode; /*声明树根指针*/
  
```

```

22     b_tree currentnode;    /*声明目前节点指针*/
23     b_tree parentnode;    /*声明父节点指针*/
24
25     /*建立新节点的内存空间*/
26     newnode=(b_tree) malloc (sizeof(treenode));
27
28     newnode->data =node;    /*存入节点内容*/
29     newnode->right=NULL;    /*设置右指针初值*/
30     newnode->left=NULL;    /*设置左指针初值*/
31
32     if (root == NULL)
33         return newnode;    /*返回新节点的位置*/
34     else
35     {
36         currentnode=root;    /*存储目前节点指针*/
37         while ( currentnode !=NULL)
38         {
39             parentnode=currentnode;    /*存储父节点指针*/
40             if (currentnode->data > node)    /*比较节点的数值大小*/
41                 currentnode=currentnode->left;    /*左子树*/
42             else
43                 currentnode=currentnode->right;    /*右子树*/
44         }
45         if (parentnode->data > node)    /*将父节点和子节点做连结*/
46             parentnode->left = newnode;    /*子节点为父节点之左子树*/
47         else
48             parentnode->right= newnode;    /*子节点为父节点之右子树*/
49     }
50     return root;    /*返回根节点之指针*/
51 }
52 /*-----*/
53 /*          建立二叉树          */
54 /*-----*/
55 b_tree create_btree(int *data,int len)
56 {
57     b_tree root=NULL;    /*根节点指针*/
58     int i;
59
60     for (i=0; i <len; i++)    /*建立树状结构*/
61         root=insert_node(root,data[i]);
62     return root;
63 }
64 /*-----*/
65 /*          二叉树前序遍历          */
66 /*-----*/
67 void preorder(b_tree point)
68 {
69     if (point!=NULL)    /*遍历的终止条件*/
70     {
71         printf("%d ",point->data);    /*处理打印节点内容*/
72         preorder(point->left);    /*处理左子树*/
73         preorder(point->right);    /*处理右子树*/
74     }
75 }
76 /*-----*/
77 /*主程序建立二叉树,并以前序遍历输出二叉树节点内容 */
78 /*-----*/
79 void main( )
80 {
81     b_tree root=NULL;    /*树根指针*/
82

```

```

83     int i, index;
84     int value;                      /*读入输入值所使用的暂存变量*/
85     int nodelist[20];              /*声明存储输入数据之数组*/
86
87     printf("\n Please input the elements of binary tree(Exit for 0):\n");
88     index=0;
89
90     /*-----读取数值存到数组中-----*/
91     scanf("%d", &value);           /*读取输入值存到变量 value*/
92
93     while (value != 0 )             /*读取尚未结束*/
94     {   nodelist[index]= value;
95         index=index+1;
96         scanf("%d",&value);
97     }
98
99     /*-----建立二叉树-----*/
100    root=create_btrees(nodelist, index);
101
102    /*-----前序遍历二叉树-----*/
103    printf(" \nThe preorder traversal result is [ ");
104    preorder(root);
105    printf("\n");
106 }

```

运行结果:

```

C:\DS>Tree_04
Please input the elements of binary tree(Exit for 0):
6 3 1 9 5 7 4 8 0

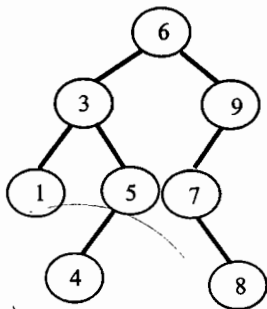
The preorder traversal result is [ 6 3 1 5 4 9 7 8 ]

C:\DS>

```

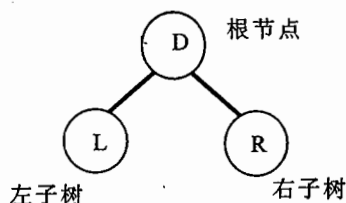
运行结果相对应的树状结构如下:

前序遍历的顺序为节点→左子树→右子树, 根节点 6 开始输出, 接着往左子树输出 3、1, 印完节点 3 之左子树, 接着输右子树 5、4。此时已完成根节点 6 之左子树, 接着处理右子树, 以同样的方式输出 7、8、9。故前序遍历的结果为: 6 3 1 5 4 9 7 8。

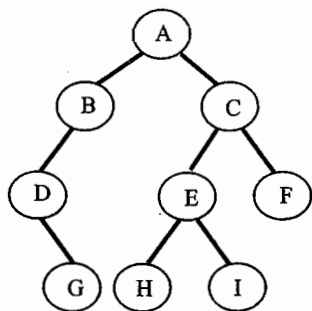


7.4.2 二叉树的中序遍历

中序遍历(Inorder traversal)是先遍历左子树, 再遍历根节点, 最后才遍历右子树。若一棵二叉树如左, 则前序遍历的顺序为: LDR, 也就是说一开始先往左方前进, 直到无法前进才处理节点, 之后再往右方前进。



例如，右图中从节点 A 开始，一直往左走到 D 无法再前进，则处理 D，再往 D 之右方到 G。此时已遍历完 B 之左子树，接着处理 B，再往 B 的右方前进。由于 B 没有右子树，故 A 之左子树遍历完毕，可处理节点 A，再往 A 之右子树前进，依此类推，故可得到中序遍历的顺序为：DGBAHEICF。



中序遍历 inorder 的步骤如下：

```

if 指向根节点的指针=NULL then
    此为空树，遍历结束
else

```

- (1) 往左走，递归处理 inorder(root→left)
- (2) 处理目前的节点
- (3) 往右走，递归处理 inorder(root→right)

程序实例：

建立二叉树，并以中序遍历的方式将节点内容输出。

程序构思：

输入元素值后建立二叉树，以递归的方式做中序遍历，其顺序为：左子树→节点→右子树，并将二叉树节点内容输出。

程序源代码：

```

01  /* ===== Program Description ===== */
02  /* 程序名称: Tree_05.c */
03  /* 程序目的: 二叉树中序遍历方式 */
04  /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05  /* ===== */
06  #include <stdlib.h>
07
08  struct tree /*声明树的结构*/
09  { struct tree *left; /*存放左子树的指针*/
10    int data; /*存放节点数据内容*/
11    struct tree *right; /*存放右子树的指针*/
12  };
13  typedef struct tree treenode; /*声明新类型树结构*/
14  typedef treenode *b_tree; /*声明二叉树的链表*/
15
16  /*-----*/
17  /* 插入二叉树的节点 */
18  /*-----*/
19  b_tree insert_node(b_tree root,int node)
20  {
21      b_tree newnode; /*声明树根指针*/
22      b_tree currentnode; /*声明目前节点指针*/
23      b_tree parentnode; /*声明父节点指针*/
24

```

```

25     /*建立新节点的内存空间*/
26     newnode=(b_tree) malloc (sizeof(treenode));
27
28     newnode->data =node;    /*存入节点内容*/
29     newnode->right=NULL;    /*设置右指针初值*/
30     newnode->left=NULL;     /*设置左指针初值*/
31
32     if (root == NULL)
33         return newnode;    /*返回新节点的位置*/
34     else
35     {
36         currentnode=root;  /*存储目前节点指针*/
37         while ( currentnode !=NULL)
38         {
39             parentnode=currentnode;    /*存储父节点指针*/
40             if (currentnode->data > node)    /*比较节点的数值大小*/
41                 currentnode=currentnode->left;    /*左子树*/
42             else
43                 currentnode=currentnode->right;    /*右子树*/
44         }
45         if (parentnode->data > node)    /*将父节点和子节点做连结*/
46             parentnode->left = newnode;    /*子节点为父节点之左子树*/
47         else
48             parentnode->right= newnode;    /*子节点为父节点之右子树*/
49     }
50     return root;    /*返回根节点之指针*/
51 }
52 /*-----*/
53 /*          建立二叉树          */
54 /*-----*/
55 b_tree create_btree(int *data,int len)
56 {
57     b_tree root=NULL;    /*根节点指针*/
58     int i;
59
60     for (i=0; i <len; i++)    /*建立树状结构*/
61         root=insert_node(root,data[i]);
62     return root;
63 }
64 /*-----*/
65 /*          二叉树中序遍历          */
66 /*-----*/
67 void inorder(b_tree point)
68 {
69     if (point!=NULL)    /*遍历的终止条件*/
70     {
71         inorder(point->left);    /*处理左子树*/
72         printf("%d ",point->data);    /*处理打印节点内容*/
73         inorder(point->right);    /*处理右子树*/
74     }
75 }
76 /*-----*/
77 /*主程序建立二叉树,并以中序遍历输出二叉树节点内容 */
78 /*-----*/
79 void main( )
80 {
81     b_tree root=NULL;    /*树根指针*/
82
83     int i,index;
84     int value;    /*读入输入值所使用的暂存变量*/
85     int nodelist[20];    /*声明存储输入数据之数组*/

```

```

86
87     printf("\n Please input the elements of binary tree(Exit for 0):\n");
88     index=0;
89
90     /*-----读取数值存到数组中-----*/
91     scanf("%d", &value);          /*读取输入值存到变量 value*/
92
93     while (value != 0 )           /*读取尚未结束*/
94     {   nodelist[index]= value;
95         index=index+1;
96         scanf("%d",&value);
97     }
98
99     /*-----建立二叉树-----*/
100    root=create_btrees(nodelist,index);
101
102    /*-----中序遍历二叉树-----*/
103    printf(" \nThe inorder traversal result is [ ");
104    inorder(root);
105    printf("\n");
106 }

```

运行结果:

```

C:\DS>Tree_05
Please input the elements of binary tree(Exit for 0):
6 3 1 9 5 7 4 8 0

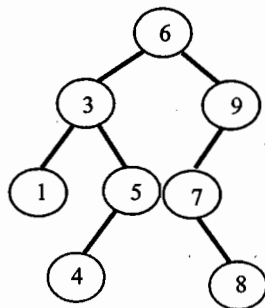
The preorder traversal result is [ 1 3 4 5 6 7 8 9 ]

C:\DS>

```

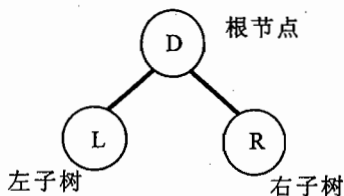
运行结果相对应的树状结构如右:

中序遍历的顺序为左子树→节点→右子树, 根节点 6 开始往左到节点 1, 输出节点 3 之左子树 1 后接着输出节点 3。再继续处理 3 之右子树 4、5。此时已完成根节点 6 之左子树, 故输出节点 6 后处理 6 之右子树, 依相同的方式输出 7、8、9。故中序遍历的结果为: 1 3 4 5 6 7 8 9。

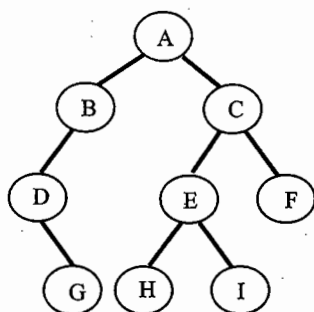


7.4.3 二叉树的后序遍历

后序遍历(Postorder traversal)是先遍历左子树, 再遍历右子树, 最后才遍历根节点。若一棵二叉树如右, 则前序遍历的顺序为: LRD, 也就是说一开始先往左方前进, 直到无法前进才再往节点的右方前进, 最后才处理节点。



例如,右图中从节点 A 开始一直往左走到 D 无法再前进,则往 D 之右方前进到 G,由于 G 没有左、右子树,故处理节点 G。之后由于 D 之右子树遍历完毕,故进而处理 D,而 B 之左子树也相对地完成。且节点 B 没有右子树,故可接着处理 B。此时节点 A 之左子树已遍历完毕,可进而往 A 之右子树 C 前进,依此类推,当 A 之右子树遍历完毕后,方可处理根节点 A,故可得到后序遍历的顺序为:G D B H I E F C A。



后序遍历的步骤如下:

- ```

if 指向根节点的指针=NULL then
 此为空树,遍历结束
else
 (1) 往左走,递归处理 postorder(root→left)
 (2) 往右走,递归处理 postorder(root→right)
 (3) 处理目前的节点

```

程序实例:

建立二叉树,并以后序遍历的方式将节点内容输出。

程序构思:

输入元素值后建立二叉树,以递归的方式做后序遍历,其顺序为:左子树→右子树→节点,并将二叉树节点内容输出。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_06.c */
03 /* 程序目的: 二叉树后序遍历方式 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07
08 struct tree /*声明树的结构*/
09 { struct tree *left; /*存放左子树的指针*/
10 int data; /*存放节点数据内容*/
11 struct tree *right; /*存放右子树的指针*/
12 };
13 typedef struct tree treetype; /*声明新类型树结构*/
14 typedef treetype *b_tree; /*声明二叉树的链表*/
15
16 /*-----*/
17 /* 插入二叉树的节点 */
18 /*-----*/
19 b_tree insert_node(b_tree root,int node)
20 {
21 b_tree newnode; /*声明树根指针*/
22 b_tree currentnode; /*声明目前节点指针*/
23 b_tree parentnode; /*声明父节点指针*/

```



```

24
25 /*建立新节点的内存空间*/
26 newnode=(b_tree) malloc (sizeof(treenode));
27
28 newnode->data =node; /*存入节点内容*/
29 newnode->right=NULL; /*设置右指针初值*/
30 newnode->left=NULL; /*设置左指针初值*/
31
32 if (root == NULL)
33 return newnode; /*返回新节点的位置*/
34 else
35 {
36 currentnode=root; /*存储目前节点指针*/
37 while (currentnode !=NULL)
38 {
39 parentnode=currentnode; /*存储父节点指针*/
40 if (currentnode->data > node) /*比较节点的数值大小*/
41 currentnode=currentnode->left; /*左子树*/
42 else
43 currentnode=currentnode->right; /*右子树*/
44 }
45 if (parentnode->data > node) /*将父节点和子节点做连结*/
46 parentnode->left = newnode; /*子节点为父节点之左子树*/
47 else
48 parentnode->right= newnode; /*子节点为父节点之右子树*/
49 }
50 return root; /*返回根节点之指针*/
51 }
52 /*-----*/
53 /* 建立二叉树 */
54 /*-----*/
55 b_tree create_btree(int *data,int len)
56 {
57 b_tree root=NULL; /*根节点指针*/
58 int i;
59
60 for (i=0; i <len; i++) /*建立树状结构*/
61 root=insert_node(root,data[i]);
62 return root;
63 }
64 /*-----*/
65 /* 二叉树后序遍历 */
66 /*-----*/
67 void postorder(b_tree point)
68 {
69 if (point!=NULL) /*遍历的终止条件*/
70 {
71 postorder(point->left); /*处理左子树*/
72 postorder(point->right); /*处理右子树*/
73 printf("%d ",point->data); /*处理打印节点内容*/
74 }
75 }
76 /*-----*/
77 /*主程序建立二叉树,并以后序遍历输出二叉树节点内容 */
78 /*-----*/
79 void main()
80 {
81 b_tree root=NULL; /*树根指针*/
82
83 int i,index;
84 int value; /*读入输入值所使用的暂存变量*/

```

```

85 int nodelist[20]; /*声明存储输入数据之数组*/
86
87 printf("\n Please input the elements of binary tree(Exit for 0):\n");
88 index=0;
89
90 /*-----读取数值存到数组中-----*/
91 scanf("%d", &value); /*读取输入值存到变量 value*/
92
93 while (value != 0) /*读取尚未结束*/
94 { nodelist[index]= value;
95 index=index+1;
96 scanf("%d",&value);
97 }
98
99 /*-----建立二叉树-----*/
100 root=create_btree(nodelist,index);
101
102 /*-----后序遍历二叉树-----*/
103 printf(" \nThe postorder traversal result is [");
104 postorder(root);
105 printf("\n");
106 }

```

运行结果:

```

C:\DS>Tree_06
Please input the elements of binary tree(Exit for 0):
6 3 1 9 5 7 4 8 0

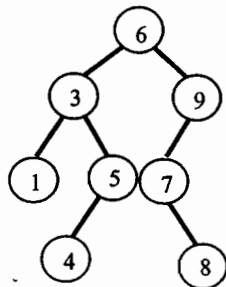
The preorder traversal result is [1 4 5 3 8 7 9 6]

C:\DS>

```

运行结果相对应的树状结构如右:

后序遍历的顺序为左子树→右子树→节点, 根节点 6 开始往左到节点 1, 输出节点 3 之左子树 1 后接着处理右子树 4、5, 再输出节点 3。此时已完成根节点 6 之左子树, 再依相同的方式处理节点 6 之右子树输出 8、7、9, 最后再输出根节点 6。故后序遍历的结果为: 1 4 5 3 8 7 9 6。



## 7.5 二叉树的建立(递归法)

在 7.3.1 节二叉树数组表示法中, 是将二叉树的节点置入适当的数组位置中, 由于如果二叉树的节点分佈不平均时, 容易造成内存空间之使用率偏低。而使用链表结构可利用指针直接指向其左、右子点, 可减少空间的浪费。本章将使用递归的方式, 将一个表示二叉树的数组结构转换成链表结构, 事实上二叉树的递归建立法和 7.4 节二叉树的遍历非常类似。

程序实例:

给定一个二叉树数组结构, 使用递归方式建立一棵二叉树, 并以中序遍历的方式输出二叉树节点内容。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_07.c */
03 /* 程序目的: 二叉树递归建立法 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07
08 struct tree /*声明树的结构*/
09 { struct tree *left; /*存放左子树的指针*/
10 int data; /*存放节点数据内容*/
11 struct tree *right; /*存放右子树的指针*/
12 };
13 typedef struct tree tnode; /*声明新类型树结构*/
14 typedef tnode *b_tree; /*声明二叉树的链表*/
15
16 /*-----*/
17 /* 使用递归建立树状结构 */
18 /*-----*/
19 b_tree create_btree(int *nodelist,int position)
20 {
21 b_tree newnode; /*声明新节点指针*/
22
23 if (nodelist[position] == 0 || position > 15) /*递归的终止条件*/
24 return NULL;
25 else
26 {
27 /*-----建立新节点的内存空间-----*/
28 newnode=(b_tree) malloc (sizeof(tnode));
29
30 /*-----建立节点内容-----*/
31 newnode->data=nodelist[position];
32
33 /*-----递归建立左子树-----*/
34 newnode->left=create_btree(nodelist,2*position);
35
36 /*-----递归建立右子树-----*/
37 newnode->right=create_btree(nodelist,2*position+1);
38
39 return newnode; /*返回复制树的位置*/
40 }
41 }
42
43 /*-----*/
44 /* 二叉树中序遍历打印节点内容 */
45 /*-----*/
46 void inorder_print_btree(b_tree point)
47 {
48 if (point!=NULL) /*遍历的终止条件*/
49 {
50 inorder_print_btree(point->left); /*处理左子树*/
51 printf("[%2d] ",point->data); /*处理打印节点内容*/
52 inorder_print_btree(point->right); /*处理右子树*/
53 }
54 }
55 /*-----*/
56 /*主程序:建立链表二叉树,并以中序遍历打印节点内容 */
57 /*-----*/
58 void main ()
59 {

```

```

60 b_tree root=NULL; /*树根指针*/
61 int i;
62 /*-----声明二叉树数组节点数据-----*/
63 int nodelist[16]={0,5,2,9,1,4,7,0,0,0,3,0,6,8,0,0};
64
65 /*-----建立树状结构-----*/
66 root=create_btree(nodelist,1);
67
68 /*-----打印原数组中的节点内容-----*/
69 printf("\nThe node content of array_structure is:\n");
70 for (i=1;i<16;i++)
71 printf("[%2d]", nodelist[i]);
72 printf("\n");
73
74 /*-----打印树状结构(链表)的节点内容-----*/
75 printf("\nThe node content of linklist_structure is:\n");
76 inorder_print_btree(root);
77 printf("\n");
78 }

```

运行结果:

```

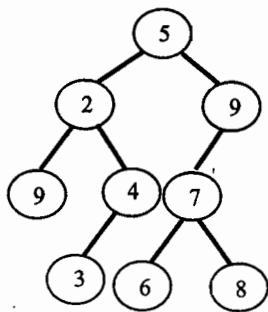
C:\DS>Tree_07
The node content of array_structure is:
[0] [5] [2] [9] [1] [4] [7] [0] [0] [0] [3] [0] [6] [8] [0] [0]

The node content of linklist_structure is:
[1] [2] [3] [4] [5] [6] [7] [8] [9]

C:\DS>

```

运行结果相对应的树状结构如下:

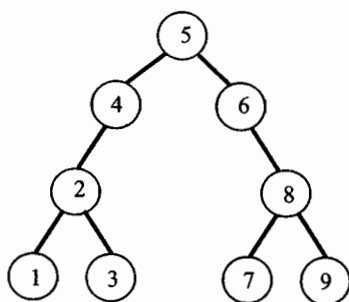


## 7.6 二叉树的查找

### 7.6.1 何谓二叉查找树

在7.3.1节中也提到建立二叉树的基本原则,事实上已满足了二叉查找树(Binary search tree)的条件:每个节点的数据要大于左子节点的数据,且要小于右子节点的数据。

例如：



以图中节点 5 来看，其数据大于左子节点 4，且小于右子节点 6，其它节点也都满足这样的条件，故此树为二叉查找树。而一般二叉树的建立基本上也都会满足相同的条件。

## 7.6.2 二叉树的查找方式

由于在二叉树中会以各节点为准将其下的元素分为左、右两部分，这样的特性使得对二叉树的查找更为容易。例如，我们欲在上图中寻找节点 9，第一步先和节点 5 比较，因为较大节点必在二叉树的左子树，继续和节点 6 比较，结果还是比节点 6 大，故继续往右前进。节点 9 和节点 8 比较，结果节点 9 较大，故往右前进达节点 9，此时找到了我们欲找得节点数据。

在这样的查找过程中，共花了 4 次比较找到了欲寻找的节点数据，若是以二叉树遍历的方式来寻找，就必须先遍历左子树，再遍历右子树，其中共需要 8 次比较次数，由此可看出这两者的执行效率有明显的差异。

在下列的程序中包含二分查找和二叉树遍历查找两种方式，读者可自行比较其差异性。

**程序实例：**

以递归的方式建立二叉树，使其满足二叉查找树。并输入欲找寻的值，分别用二叉树遍历及二分查找方式进行查找，最后输出查找结果。

**程序构思：**

若使用二分查找的方式必须要是二叉查找树，故在建立二叉树时要满足二叉查找树的条件（左小右大），在进行查找时若值较节点小则使用递归往左子树继续找（btree\_travesal\_search(point->left,findnode)），若值较节点大也使用递归往右子树继续找（使用 btree\_travesal\_search(point->right,findnode)），最后再输出查找结果。

**程序源代码：**

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_08.c */
03 /* 程序目的: 二叉树的查找 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07
08 struct tree /*声明树的结构*/
09 { struct tree *left; /*存放左子树的指针*/
10 int data; /*存放节点数据内容*/
11 struct tree *right; /*存放右子树的指针*/
12 };

```

```

13 typedef struct tree treenode; /*声明新类型树结构*/
14 typedef treenode *b_tree; /*声明二叉树的链表*/
15 /*-----*/
16 /* 使用递归建立二叉树 */
17 /*-----*/
18 b_tree create_bt看ree(int *nodelist,int position)
19 {
20 b_tree newnode; /*声明新节点指针*/
21
22 if (nodelist[position] == 0 || position > 15) /*递归的终止条件*/
23 return NULL;
24 else
25 {
26 /*-----建立新节点的内存空间-----*/
27 newnode=(b_tree) malloc (sizeof(treenode));
28
29 /*-----建立节点内容-----*/
30 newnode->data=nodelist[position];
31
32 /*-----递归建立左子树-----*/
33 newnode->left=create_btree(nodelist,2*position);
34 /*-----递归建立右子树-----*/
35 newnode->right=create_btree(nodelist,2*position+1);
36
37 return newnode; /*返回复制树的位置*/
38 }
39 }
40
41 /*-----*/
42 /* 二叉树遍历查找方式 */
43 /*-----*/
44 b_tree btree_travesal_search(b_tree point,int findnode)
45 {
46 b_tree point1,point2; /*声明往左及往右查找的指针*/
47
48 if (point!=NULL) /*递归的终止条件*/
49 {
50 if (point->data == findnode) /*找到了欲寻找的节点*/
51 return point; /*返回找到节点的指针*/
52 else
53 /*找左子树*/
54 point1=btree_travesal_search(point->left,findnode);
55 /*找右子树*/
56 point2=btree_travesal_search(point->right,findnode);
57
58 if (point1 != NULL)
59 return point1; /*该节点在左子树*/
60 else
61 if (point2 != NULL)
62 return point2; /*该节点在右子树*/
63 else
64 return NULL; /*该节点不在此二叉树中*/
65 }
66 else
67 return NULL;
68 }
69 /*-----*/
70 /* 二叉树二分查找方式 */
71 /*-----*/
72 b_tree btree_traversal_search(b_tree point,int findnode)
73 {

```

```

74 while (point!=NULL)
75 {
76 if (point->data == findnode) /*找到了欲寻找的节点*/
77 return point; /*返回找到节点的指针*/
78 else
79 if (point->data > findnode)
80 point=point->left; /*往左子树找*/
81 else
82 point=point->right; /*往右子树找*/
83 }
84 return NULL; /*该节点不在此二叉树中*/
85 }
86
87 /*-----*/
88 /* 主程序：二叉树的查找 */
89 /*-----*/
90 void main ()
91 {
92 b_tree root=NULL; /*声明树根指针*/
93 b_tree point=NULL; /*声明节点指针*/
94 int findnode; /*欲寻找的节点数据值*/
95
96 /*-----声明二叉树数组节点数据-----*/
97 int nodelist[16]={0,5,2,9,0,4,7,0,0,0,3,0,6,8,0,0};
98
99 /*-----建立树状结构-----*/
100 root=create_btrees(nodelist,1);
101
102 /*-----读取欲寻找的节点数据-----*/
103 printf("\nPlease input the node value(1.9) you want search:");
104 scanf("%d",&findnode);
105
106 /*-----进行遍历查找-----*/
107 point= btrees_traversal_search(root,findnode);
108 if (point!=NULL)
109 {
110 printf("\n=>Traversal search result: \n");
111 printf(" The finding node value is [%d]\n",point->data);
112 }
113 else
114 printf("\nTraversal search result: Not find!!\n");
115
116 /*-----进行二分查找-----*/
117 point= btrees_traversal_search(root,findnode);
118 if (point!=NULL)
119 {
120 printf("\n=>Binary search result: \n");
121 printf(" The finding node value is [%d]\n",point->data);
122 }
123 else
124 printf("\nBinary search result: Not find!!\n");
125 }

```

运行结果:

```

C:\DS>Tree_08
Please input the node value(1.9) you want search: 1

=>Traversal search result: Not find!!
=>Binary search result: Not find!!

```

```
Please input the node value(1.9) you want search: 3
```

```
=>Traversal search result:
The finding node value is [3]
```

```
=>Binary search result:
The finding node value is [3]
```

```
C:\DS>
```

## 7.7 二叉树的节点删除

对于一个二叉树，若欲删除其节点，应先寻找欲删除的节点是否存在于该二叉树中。关于二叉树的节点查找，在7.6节已有详细的介绍，本章节将说明如何将节点从二叉树中删除。

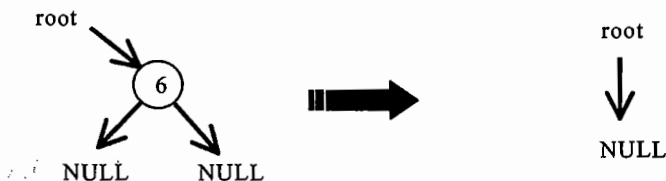
由于我们在删除一个节点后，必须要维持满足二叉查找树数据排列的原则：左子节点<节点<右子节点。而删除节点的处理可分4种情况，我们将对各种情况做详细的说明。

### 7.7.1 节点无左子树，无右子树

当欲删除一无左子树也无右子树的节点时，需要考虑到两种情况：

#### (1) 为根节点

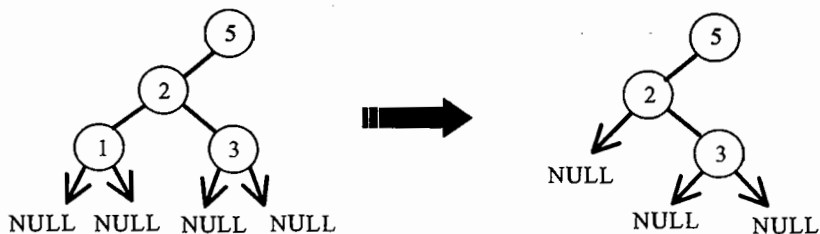
如欲删除无左、右子树的根节点，只需将根节点指针 root 指向 NULL 即可。



#### (2) 非根节点

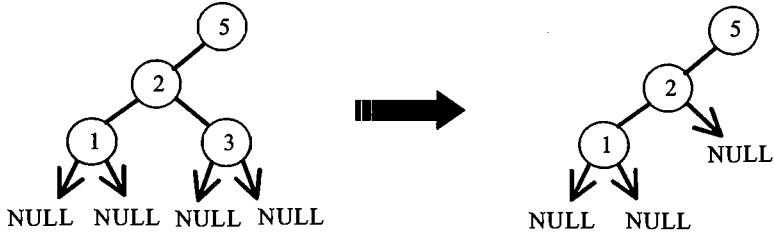
若一节点为无左、右子树的非根节点，那么该节点必为叶节点。如果节点为父节点的左子节点，则将父节点的左指针指向 NULL，相同地，若节点为父节点的右子节点，则将父节点的右指针指向 NULL。

例如删除图中节点1(为父节点的左子节点)，则将节点2的左指针指向 NULL。





若删除图中节点 3(为父节点 2 的右子节点), 则将节点 2 的右指针指向 NULL。

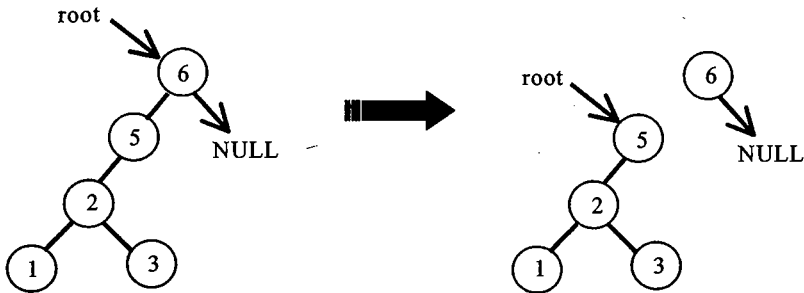


### 7.7.2 节点有左子树, 无右子树

当欲删除一有左子树但无右子树的节点时, 也需去考虑两种情况:

(1) 为根节点

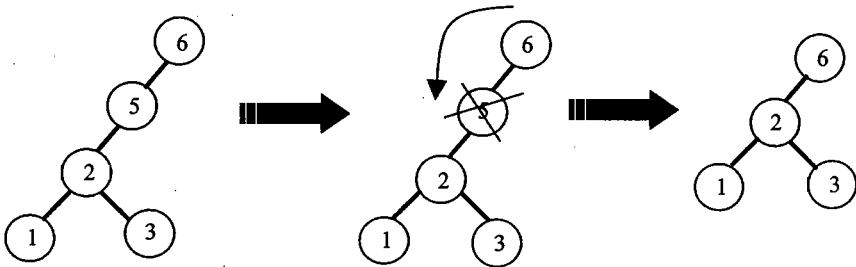
如欲删除有左子树, 无右子树之根节点 6, 只需将根节点指针 root 指向其左子树。



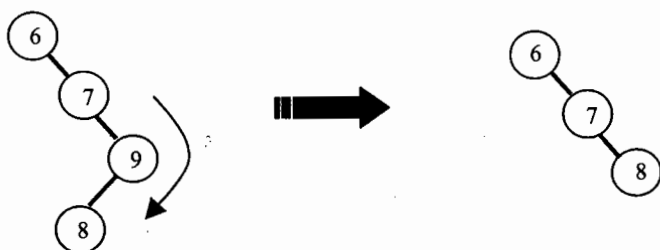
(2) 非根节点

一节点为左子树, 无右子树的非根节点, 若节点为父节点的左子节点, 则将父节点的左指针指向节点的左子节点, 相同地, 若节点为父节点的右子节点, 则将父节点的右指针指向节点的左子节点。

例如删除图中的节点 5(为父节点 6 的左子节点), 则将节点 6 的左指针指向节点 5 的左子节点 2。

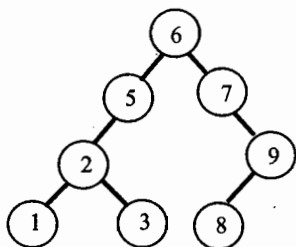


若删除图中节点 9(为父节点 7 的右子节点), 则将节点 7 的右指针指向节点 9 的左子节点 8。



### 7.7.3 节点无左子 100 树，有右子树

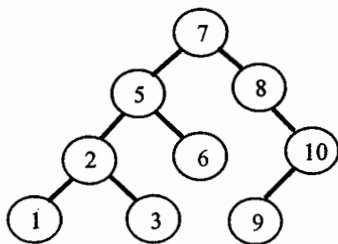
如图中节点 7。



欲删除一无左子树、有右子树的节点其方法和 7.7.2 节的做法类似，留给读者自行练习。

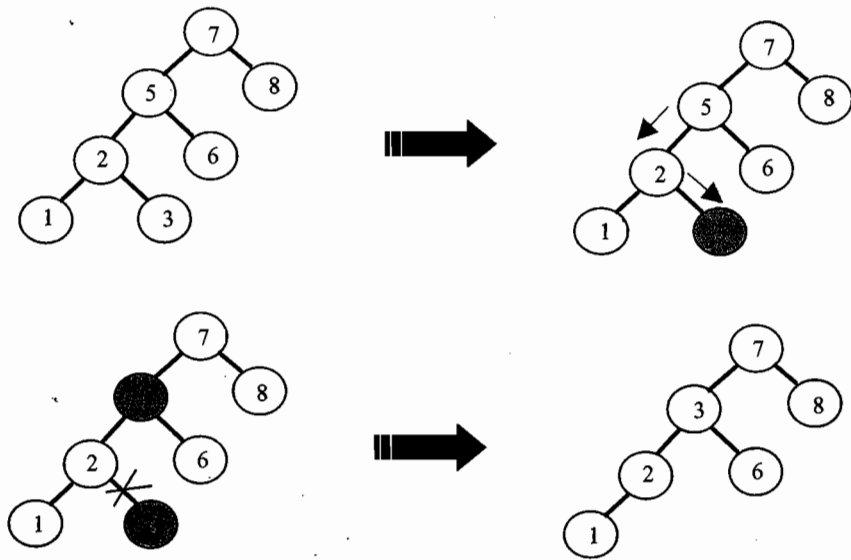
### 7.7.4 节点有左子树，有右子树

如图中节点 5、7。



当欲删除一具有左子树也具有右子树的节点时，不需要考虑 7.7.2 节和 7.7.3 节的两种情况，但删除的过程较复杂。由于节点具有左、右子树，故删除节点后需找一个替代的节点值，以免除大量地移动节点。而寻找替代节点的方法有两种：一为找节点左子树的最右边的点，二为找节点右子树的最左边的点，本文采第一种方法。当找到左子树的最右边节点 R 时，将欲删除的节点值替换成节点 R 值，并释放原节点 R 的内存空间。

例如删除图中节点 5，先找到其左子树之最右边节点 3，将 3 取代掉节点 5，再释放节点 3 的内存空间。



下面的范例为一处理二叉树节点删除的完整程序。

#### 程序实例：

输入一节点，并从二叉树中删除该节点。

#### 程序构思：

进行节点删除时要先判断其所属位置，用变量 `position` 来记录，另外在查找节点的过程中需要保留父节点指针 `parent`，以进行删除时的节点链接。

节点的删除根据上述各种情况做处理，完成删除后使用函数 `free()` 释放空间。

#### 程序源代码：

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_09.c */
03 /* 程序目的: 删除二叉树中的节点 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07
08 struct tree /*声明树的结构*/
09 { struct tree *left; /*存放左子树的指针*/
10 int data; /*存放节点数据内容*/
11 struct tree *right; /*存放右子树的指针*/
12 };
13 typedef struct tree treenode; /*声明新类型树结构*/
14 typedef treenode *b_tree; /*声明二叉树的链表*/
15 /*-----*/
16 /* 使用递归建立二叉树 */
17 /*-----*/
18 b_tree create_btree(int *nodelist,int position)
19 {
20 b_tree newnode; /*声明新节点指针*/

```

```

21
22 if (nodelist[position] == 0 || position>15)/*递归的终止条件*/
23 return NULL;
24 else
25 {
26 /*-----建立新节点的内存空间-----*/
27 newnode=(b_tree) malloc (sizeof(treenode));
28 /*-----建立节点内容-----*/
29 newnode->data=nodelist[position];
30
31 /*-----递归建立左子树-----*/
32 newnode->left=create_btrees(nodelist,2*position);
33
34 /*-----递归建立右子树-----*/
35 newnode->right=create_btrees(nodelist,2*position+1);
36
37 return newnode; /*返回新节点的位置*/
38 }
39 }
40 /*-----*/
41 /* 二叉树二分查找方式 */
42 /*-----*/
43 b_tree btrees_traversal_search(b_tree point,int findnode)
44 {
45 while (point!=NULL)
46 {
47 if (point->data == findnode) /*找到了欲寻找之节点*/
48 return point; /*返回找到节点之指针*/
49 else if (point->data > findnode)
50 point=point->left; /*往左子树找*/
51 else
52 point=point->right; /*往右子树找*/
53 }
54 return NULL; /*该节点不在此二叉树中*/
55 }
56 /*-----*/
57 /* 查找节点是否在二叉树中 */
58 /*-----*/
59 b_tree binary_search(b_tree point,int node,int *position)
60 {
61 b_tree parent; /*声明父节点的指针*/
62
63 parent=point; /*设置指针初始值*/
64 *position=0; /*设置位置的初始值*/
65 while (point !=NULL)
66 {
67 if (point->data == node) /*找到该节点*/
68 return parent;
69 else
70 {
71 parent=point; /*保留父节点指针*/
72 if (point->data > node) /*比较数据*/
73 {
74 point=point->left; /*指向左子树*/
75 *position=-1; /*设置其为左子树时 position 为-1*/
76 }
77 else
78 {
79 point=point->right; /*指向右子树*/
80 *position=1; /*设置其为右子树时 position 为 1*/
81 }

```

```

82 }
83 }
84 return NULL;
85 }
86 /*-----*/
87 /* 进行节点的删除 */
88 /*-----*/
89 b_tree delete_node(b_tree root,int node)
90 {
91 b_tree parent; /*父节点指针*/
92 b_tree point; /*欲删除节点之指针*/
93 b_tree child; /*子节点指针*/
94 int position; /*删除之位置(是左子树或右子树)*/
95
96 /*寻找欲删除之节点的父节点指针*/
97 parent= binary_search(root,node,&position);
98 if (parent==NULL) /*该节点不在二叉树中*/
99 return root;
100 else
101 {
102 switch (position) /*判断删除的位置*/
103 {
104 case -1: point=parent->left; /*左子节点*/
105 break;
106 case 1: point=parent->right; /*右子节点*/
107 break;
108 case 0: point=parent; /*根节点*/
109 break;
110 }
111 /*-----没有左子树也没有右子树-----*/
112 if (point->left==NULL && point->right==NULL)
113 {
114 switch (position) /*判断删除的位置*/
115 {
116 case -1: parent->left=NULL; /*将父节点的右指针指向 NULL*/
117 break;
118 case 1: parent->right=NULL; /*将父节点的左指针指向 NULL*/
119 break;
120 case 0: parent=NULL; /*根节点指向 NULL*/
121 break;
122 }
123 return root;
124 }
125 /*-----没有左子树-----*/
126 if (point->left==NULL && point->right!=NULL)
127 {
128 if (parent!=point)
129 /*将父节点的右指针指向节点的右子节点*/
130 parent->right=point->right;
131 else
132 root=root->right; /*将根节点指向右子节点*/
133 free(point); /*释放节点内存*/
134 return root; /*返回根节点指针*/
135 }
136 /*-----没有右子树-----*/
137 else if (point->right==NULL && point->left!=NULL)
138 {
139 if (parent!=point)
140 /*将父节点的左指针指向节点的左子节点*/
141 parent->left=point->left;
142 else

```

```

143 root=root->left; /*将根节点指向左子节点*/
144 free(point); /*释放节点内存*/
145 return root; /*返回根节点指针*/
146 }
147
148 /*-----有左子树也有右子树-----*/
149 else if (point->right!=NULL && point->left!=NULL)
150 {
151 parent=point; /*将父节点指向目前的节点*/
152 child=point->left; /*设置子节点*/
153 /*找到左子树中之最右边的叶子点*/
154 while (child->right !=NULL)
155 {
156 parent=child; /*保留父节点的指针*/
157 child=child->right; /*往右子树前进*/
158 }
159 /*将原节点设置成叶节点的数据值*/
160 point->data=child->data;
161 if (parent->left == child)
162 parent->left=child->left;
163 else
164 parent->right=child->right;
165 free(child); /*释放节点内存*/
166 return root; /*返回根节点指针*/
167 }
168 }
169
170 /*-----*/
171 /* 中序遍历打印二叉树 */
172 /*-----*/
173 void inorder_btree(b_tree point)
174 {
175 if (point != NULL) /*递归的终止条件*/
176 {
177 inorder_btree(point->left);
178 printf("[%2d] ",point->data);
179 inorder_btree(point->right);
180 }
181 }
182 /*-----*/
183 /*主程序:递归建立二叉树,并删除一二叉树的节点*/
184 /*-----*/
185 void main ()
186 {
187 b_tree root=NULL; /*声明树根指针*/
188 int deletenode; /*欲删除之节点数据值*/
189
190 /*-----声明二叉树数组节点数据-----*/
191 int nodelist[16]={0,5,4,6,2,0,0,8,1,3,0,0,0,0,7,9};
192
193 /*-----建立树状结构-----*/
194 root=create_btree(nodelist,1);
195
196 printf("\nThe original tree is ");
197 inorder_btree(root);
198 printf("\n");
199
200 printf("\nPlease input the node you want to delete:");
201 scanf("%d",&deletenode);
202 root=delete_node(root,deletenode); /*删除节点*/
203 printf("\nThe deleted tree is ");

```

```

204 inorder_btree(root);
205 printf("\n");
206 }

```

运行结果:

```

C:\DS>Tree_09
The original tree is [1][2][3][4][5][6][7][8][9]
Please input the node you want to delete:1
The deleted tree is [2][3][4][5][6][7][8][9]

The original tree is [1][2][3][4][5][6][7][8][9]
Please input the node you want to delete:5
The deleted tree is [1][2][3][4][6][7][8][9]

C:\DS>

```

## 7.8 二叉树的复制

在二叉树的应用中常需要将原本的二叉树备份起来,故本节将以递归的方式来复制一棵二叉树,其中复制部分的步骤有3:

- (1) 复制节点的数据内容
- (2) 复制左子树
- (3) 复制右子树

其中第2、3步骤均用递归的方式处理,且此两步骤顺序颠倒并不会影响复制的结果。

程序实例:

使用递归方式建立二叉树,再复制原来的二叉树,并输出原本的二叉树和备份二叉树的节点内容。

程序构思:

建立完二叉树后,使用一个递归函数 `copy_btree()` 来复制另一棵一模一样的二叉树,先复制节点的数据内容,接着递归复制原节点左子树到复制树的右子树 `newnode->left=copy_btree(root->left)`,再递归复制原右子树到复制树的右子树 `newnode->right=copy_btree(root->right)`,如此则复制完成。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_10.c */
03 /* 程序目的: 二叉树的复制 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07
08 struct tree /*声明树的结构*/
09 { struct tree *left; /*存放左子树的指针*/
10 int data; /*存放节点数据内容*/
11 struct tree *right; /*存放右子树的指针*/
12 };
13 typedef struct tree treenode; /*声明新类型树结构*/
14 typedef treenode *b_tree; /*声明二叉树的链表*/

```

```

15 /*-----*/
16 /* 使用递归建立二叉树 */
17 /*-----*/
18 b_tree create_btree(int *nodelist,int position)
19 {
20 b_tree newnode; /*声明新节点指针*/
21
22 if (nodelist[position] == 0 || position > 15) /*递归的终止条件*/
23 return NULL;
24 else
25 {
26 /*-----建立新节点的内存空间-----*/
27 newnode=(b_tree) malloc (sizeof(treenode));
28
29 /*-----建立节点内容-----*/
30 newnode->data=nodelist[position];
31
32 /*-----递归建立左子树-----*/
33 newnode->left=create_btree(nodelist,2*position);
34
35 /*-----递归建立右子树-----*/
36 newnode->right=create_btree(nodelist,2*position+1);
37
38 return newnode; /*返回复制树的位置*/
39 }
40 }
41 /*-----*/
42 /* 复制二叉树 */
43 /*-----*/
44 b_tree copy_btree(b_tree root)
45 {
46 b_tree newnode; /*声明新节点指针*/
47
48 if (root == NULL) /*递归的终止条件*/
49 return NULL;
50 else
51 {
52 /*-----建立新节点的内存空间-----*/
53 newnode=(b_tree) malloc (sizeof(treenode));
54
55 /*-----建立节点内容-----*/
56 newnode->data=root->data;
57
58 /*-----递归建立左子树-----*/
59 newnode->left=copy_btree(root->left);
60
61 /*-----递归建立右子树-----*/
62 newnode->right=copy_btree(root->right);
63
64 return newnode; /*返回复制树的位置*/
65 }
66 }
67 /*-----*/
68 /* 二叉树中序遍历打印节点内容 */
69 /*-----*/
70 void inorder_print_btree(b_tree point)
71 {
72 if (point!=NULL) /*遍历的终止条件*/
73 {
74 inorder_print_btree(point->left); /*处理左子树*/
75 printf("[%2d]",point->data); /*处理打印节点内容*/

```



```

76 inorder_print_btree(point->right); /*处理右子树*/
77 }
78 }
79 /*-----*/
80 /*主程序:建立二叉树,复制出一相同的二叉树,并输出两树之节点内容 */
81 /*-----*/
82 void main()
83 {
84 b_tree root=NULL; /*声明原二叉树指针*/
85 b_tree copytree=NULL; /*声明复制二叉树指针*/
86
87 /*-----声明二叉树数组节点数据-----*/
88 int nodelist[16]={0,5,2,9,0,4,7,0,0,0,3,0,6,8,0,0};
89
90 /*-----建立树状结构-----*/
91 root=create_btree(nodelist,1);
92
93 /*-----复制二叉树-----*/
94 copytree=copy_btree(root);
95
96 /*-----打印原二叉树节点内容-----*/
97 printf("\nOriginal binary tree node content:\n");
98 inorder_print_btree(root);
99 printf("\n");
100
101 /*-----打印备份二叉树节点内容-----*/
102 printf("\nBackup binary tree node content:\n");
103 inorder_print_btree(copytree);
104 printf("\n");
105 }

```

运行结果:

```

C:\DS>Tree_10
Original binary tree node content:
[2] [3] [4] [5] [6] [7] [8] [9]

Backup binary tree node content:
[2] [3] [4] [5] [6] [7] [8] [9]

C:\DS>

```

## 7.9 二叉树的比较

当我们欲判断两棵二叉树是否相等时,需要进行二叉树的比较。

程序实例:

输入两二叉树之数据,分别建立二叉树并比较两者是否相同,最后输出比较结果及二叉树节点数据内容。

程序构思:

若欲比较之两树均为空,则两树相同,return 值为 1。若两树之根节点相同则进一步使用递归函

数 `equal_btree()` 比较其子树是否相同。若所有节点均相同则 `return` 值 1, 否则 `return` 值 0 表示两树并不相同。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_11.c */
03 /* 程序目的: 二叉树的比较 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 struct tree /*声明树的结构*/
08 { struct tree *left; /*存放左子树的指针*/
09 int data; /*存放节点数据内容*/
10 struct tree *right; /*存放右子树的指针*/
11 };
12 typedef struct tree treenode; /*声明新类型树结构*/
13 typedef treenode *b_tree; /*声明二叉树的链表*/
14 /*-----*/
15 /* 插入二叉树的节点 */
16 /*-----*/
17 b_tree insert_node(b_tree root,int node)
18 {
19 b_tree newnode; /*声明树根指针*/
20 b_tree currentnode; /*声明目前节点指针*/
21 b_tree parentnode; /*声明父节点指针*/
22
23 /*建立新节点的内存空间*/
24 newnode=(b_tree) malloc (sizeof(treenode));
25
26 newnode->data =node; /*存入节点内容*/
27 newnode->right=NULL; /*设置右指针初值*/
28 newnode->left=NULL; /*设置左指针初值*/
29
30 if (root == NULL)
31 return newnode; /*返回新节点的位置*/
32 else
33 {
34 currentnode=root; /*存储目前节点指针*/
35 while (currentnode !=NULL)
36 {
37 parentnode=currentnode; /*存储父节点指针*/
38 if (currentnode->data > node) /*比较节点的数值大小*/
39 currentnode=currentnode->left; /*左子树*/
40 else
41 currentnode=currentnode->right; /*右子树*/
42 }
43 if (parentnode->data > node) /*将父节点和子节点做连结*/
44 parentnode->left = newnode; /*子节点为父节点之左子树*/
45 else
46 parentnode->right= newnode; /*子节点为父节点之右子树*/
47 }
48 return root; /*返回根节点之指针*/
49 }
50 /*-----*/
51 /* 建立二叉树 */
52 /*-----*/
53 b_tree create_btree(int *data,int len)
54 {
55 b_tree root=NULL; /*根节点指针*/

```

```

56 int i;
57
58 for (i=0; i < len; i++) /*建立树状结构*/
59 root=insert_node(root,data[i]);
60 return root;
61 }
62
63 /*-----*/
64 /* 二叉树前序遍历 */
65 /*-----*/
66 void preorder(b_tree point)
67 {
68 if (point!=NULL) /*遍历的终止条件*/
69 {
70 printf("%d ",point->data); /*处理打印节点内容*/
71 preorder(point->left); /*处理左子树*/
72 preorder(point->right); /*处理右子树*/
73 }
74 }
75 /*-----*/
76 /* 二叉树的比较 */
77 /*-----*/
78 int equal_btree(b_tree root1,b_tree root2)
79 {
80 if (root1==NULL || root2==NULL) /*两树均为空树*/
81 return 1;
82 else if (root1->data == root2->data) /*节点数据相等*/
83 if (equal_btree(root1->left,root2->left) ==1) /*判断左子树*/
84 return equal_btree(root1->right,root2->right); /*判断右子树*/
85 else
86 return 0;
87 }
88 /*-----*/
89 /*主程序:建立两二叉树,并比较两者是否相同 */
90 /*-----*/
91 void main()
92 {
93 b_tree root1=NULL; /*树根指针*/
94 b_tree root2=NULL; /*树根指针*/
95
96 int i,index1,index2;
97 int equal; /*两树之比较结果*/
98 int value; /*读入输入值所使用的暂存变量*/
99 int nodelist1[20],nodelist2[20]; /*声明存储输入数据之数组*/
100
101 index1=0;
102 index2=0;
103
104 /*-----读取 tree(1) 数值存到数组中-----*/
105 printf("\n Please input the elements of binary tree(1) (Exit for 0):\n");
106 scanf("%d", &value); /*读取输入值存到变量 value*/
107 while (value != 0) /*读取尚未结束*/
108 {
109 nodelist1[index1]= value; /*将元素一一存入数组 nodelist1 中*/
110 index1=index1+1;
111 scanf("%d",&value);
112 }
113 /*-----读取 tree(2) 数值存到数组中-----*/
114 printf("\n Please input the elements of binary tree(2) (Exit for 0):\n");
115 scanf("%d", &value); /*读取输入值存到变量 value*/
116 while (value != 0) /*读取尚未结束*/
117 {
118 nodelist2[index2]= value; /*将元素一一存入数组 nodelist2 中*/

```

```

117 index2=index2+1;
118 scanf("%d",&value);
119 }
120 /*-----建立二叉树-----*/
121 root1=create_btree(nodelist1,index1);
122 root2=create_btree(nodelist2,index2);
123 /*-----比较两二叉树-----*/
124 equal=equal_btree(root1,root2);
125
126 printf("\nThe compare result ==>");
127 if (equal == 1)
128 printf(" Equal!!!\n");
129 else
130 printf(" Not Equal!!!\n");
131
132 /*-----前序遍历打印二叉树-----*/
133 printf("The preorder traversal of tree(1) is [");
134 preorder(root1);
135 printf("]\n");
136
137 printf("The preorder traversal of tree(2) is [");
138 preorder(root2);
139 printf("]\n");
140 }

```

运行结果:

```

C:\DS>Tree_11
Please input the elements of binary tree(1)(Exit for 0):
4 7 2 9 5 1 0
Please input the elements of binary tree(2)(Exit for 0):
4 1 7 2 5 9 0

The compare result ==> Not Equal!!!
The preorder traversal of tree(1) is [4 2 1 7 5 9]
The preorder traversal of tree(2) is [4 1 2 7 5 9]

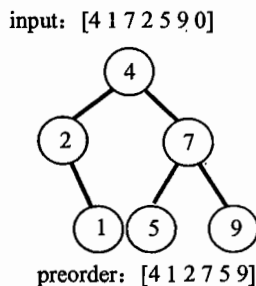
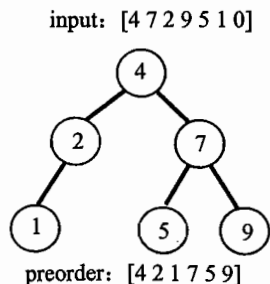
Please input the elements of binary tree(1)(Exit for 0):
4 7 2 9 5 1 0
Please input the elements of binary tree(2)(Exit for 0):
4 2 1 7 5 9 0

The compare result ==> Equal!!!
The preorder traversal of tree(1) is [4 2 1 7 5 9]
The preorder traversal of tree(2) is [4 2 1 7 5 9]

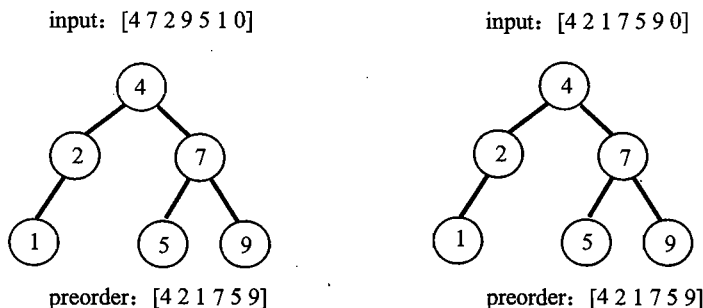
C:\DS>

```

运行 1 结果如下: 两树不相等



运行 2 结果如下：两树相等



## 7.10 二叉树的映像

另外有一种较特别的二叉树操作，称为“二叉树的映像”。顾名思义，所谓“映像”就像照镜子一般，会造成左右颠倒的状况。故“二叉树的映像”即是将二叉树左右反转，原来在左子树的变成右子树，当然，原来在右子树的就变成了左子树。

程序实例：

输入二叉树之节点数据内容建立一二叉树,进行该二叉树的映像,并输出映像前后之二叉树节点内容。

程序构思：

二叉树的映像是要将原左子节点变成右子节点，而右子节点则变成左子节点，本程序对于节点的处理也是使用递归的方式进行映像。根节点数据不变，再使用递归函数 `swap_btree()` 进行映像。将右子树变左子树是 `swaptree->left=swap_btree(root->right)`，而左子树变右子树则是 `swaptree->right=swap_btree(root->left)`，最后返回映设数之指针。若根节点原来是指向 `NULL`，则不需进行映像，直接返回 `NULL` 值。

程序源代码：

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_12.c */
03 /* 程序目的: 二叉树的映像 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07
08 struct tree /*声明树的结构*/
09 { struct tree *left; /*存放左子树的指针*/
10 int data; /*存放节点数据内容*/
11 struct tree *right; /*存放右子树的指针*/
12 };
13 typedef struct tree tnode; /*声明新类型树结构*/
14 typedef tnode *b_tree; /*声明二叉树的链表*/
15
16 /*-----*/

```

```

17 /* 插入二叉树的节点 */
18 /*-----*/
19 b_tree insert_node(b_tree root,int node)
20 {
21 b_tree newnode; /*声明树根指针*/
22 b_tree currentnode; /*声明目前节点指针*/
23 b_tree parentnode; /*声明父节点指针*/
24
25 /*建立新节点的内存空间*/
26 newnode=(b_tree) malloc (sizeof(treenode));
27
28 newnode->data =node; /*存入节点内容*/
29 newnode->right=NULL; /*设置右指针初值*/
30 newnode->left=NULL; /*设置左指针初值*/
31
32 if (root == NULL)
33 return newnode; /*返回新节点的位置*/
34 else
35 {
36 currentnode=root; /*存储目前节点指针*/
37 while (currentnode !=NULL)
38 {
39 parentnode=currentnode; /*存储父节点指针*/
40 if (currentnode->data > node) /*比较节点的数值大小*/
41 currentnode=currentnode->left; /*左子树*/
42 else
43 currentnode=currentnode->right; /*右子树*/
44 }
45 if (parentnode->data > node) /*将父节点和子节点做连结*/
46 parentnode->left = newnode; /*子节点为父节点的左子树*/
47 else
48 parentnode->right= newnode; /*子节点为父节点的右子树*/
49 }
50 return root; /*返回根节点的指针*/
51 }
52 /*-----*/
53 /* 建立二叉树 */
54 /*-----*/
55 b_tree create_btree(int *data,int len)
56 {
57 b_tree root=NULL; /*根节点指针*/
58 int i;
59
60 for (i=0; i <len; i++) /*建立树状结构*/
61 root=insert_node(root,data[i]);
62 return root;
63 }
64
65 /*-----*/
66 /* 进行二叉树映像 */
67 /*-----*/
68 b_tree swap_btree(b_tree root)
69 {
70 b_tree swaptree; /*进行映像的指针*/
71
72 if (root!=NULL)
73 {
74 swaptree=(b_tree) malloc (sizeof (treenode));
75 swaptree->data=root->data;
76 swaptree->left=swap_btree(root->right); /*右子树变左子树*/
77 swaptree->right=swap_btree(root->left); /*左子树变右子树*/

```

```

78 return swaptree;
79 }
80 else
81 return NULL;
82 }
83 /*-----*/
84 /* 二叉树中序遍历 */
85 /*-----*/
86 void inorder(b_tree point)
87 {
88 if (point!=NULL) /*遍历的终止条件*/
89 {
90 inorder(point->left); /*处理左子树*/
91 printf("%d ",point->data); /*处理打印节点内容*/
92 inorder(point->right); /*处理右子树*/
93 }
94 }
95
96 /*-----*/
97 /*主程序:建立二叉树,进行映像以中序遍历输出二叉树节点内容 */
98 /*-----*/
99 void main()
100 {
101 b_tree root=NULL; /*原根节点指针*/
102 b_tree root2=NULL;
103
104 int i,index;
105 int value; /*读入输入值所使用的暂存变量*/
106 int nodelist[20]; /*声明存储输入数据之数组*/
107
108 printf("\n Please input the elements of binary tree(Exit for 0):\n");
109 index=0;
110
111 /*-----读取数值存到数组中-----*/
112 scanf("%d", &value); /*读取输入值存到变量value*/
113
114 while (value != 0) /*读取尚未结束*/
115 {
116 nodelist[index]= value;
117 index=index+1;
118 scanf("%d",&value);
119 }
120
121 /*-----建立二叉树-----*/
122 root=create_btree(nodelist,index);
123 printf(" \nThe inorder traversal of original tree is [");
124 inorder(root);
125 printf("]\n");
126
127 /*-----进行映像-----*/
128 root2=swap_btree(root);
129 printf(" \nThe inorder traversal of swaping tree is [");
130 inorder(root2);
131 printf("]\n");
132 }

```

运行结果:

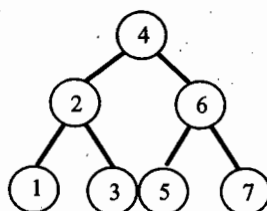
```
C:\DS>Tree_12
Please input the elements of binary tree(Exit for 0):
4 6 2 5 3 1 7 0

The inorder traversal of original tree is [1 2 3 4 5 6 7]

The inorder traversal of swaping tree is [7 6 5 4 3 2 1]

C:\DS>
```

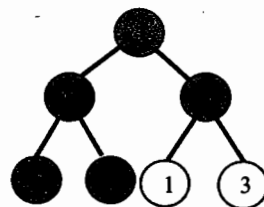
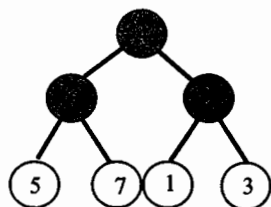
程序执行如下:



inorder [1 2 3 4 5 6 7]

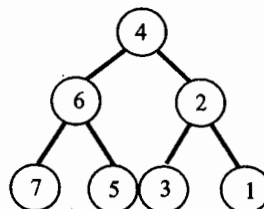
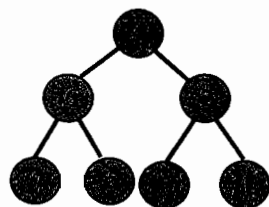
1. 对换节点4的左、右子树

2. 对换节点6有左、右子树



3. 对换节点2的左、右子树

4. 完成二叉树映像

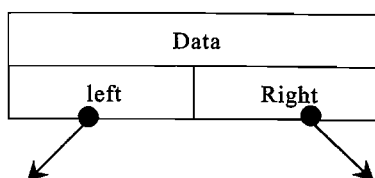


inorder [7 6 5 4 3 2 1]

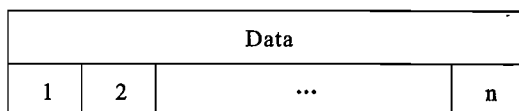
## 7.11 一般树转二叉树

在前面各章节中已介绍过二叉树的表示法及基本操作。由于二叉树之分支度最多为2，其结构如下：





相对地, 若考虑  $n$  元树, 每个节点分支度为  $n$ , 则结构如下:

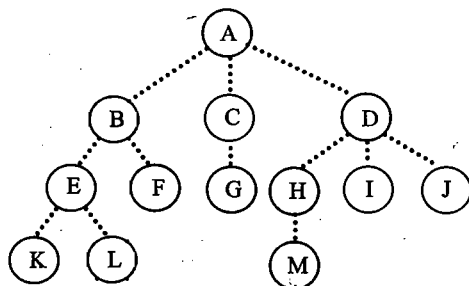


由于这样的结构相对于二叉树来说, 更容易造成空指针的问题, 故我们希望能够将一般树转换成二叉树后, 再进行各种操作处理。本节将依每个转换步骤进行说明, 程序部分留给读者自行练习。

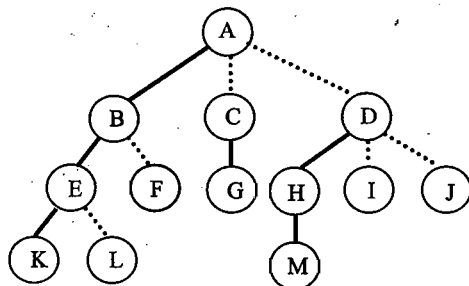
欲将一般树转换成二叉树, 也就是要将  $n$  个分支度变成 2 个分支, 主要有 4 个步骤:

- (1) 保留所有节点与其左子节点的链接
- (2) 连结所有兄弟节点(拥有同一个父节点的子节点)
- (3) 打断所有节点原本与右子节点的链接
- (4) 将兄弟节点顺时针转  $45^\circ$

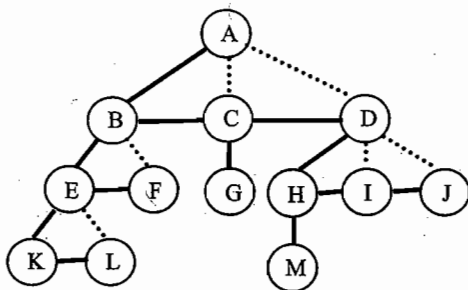
例如, 欲将下图转换成二叉树:



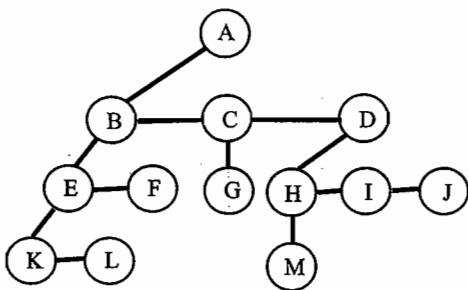
步骤 1: 保留与左子节点的链接



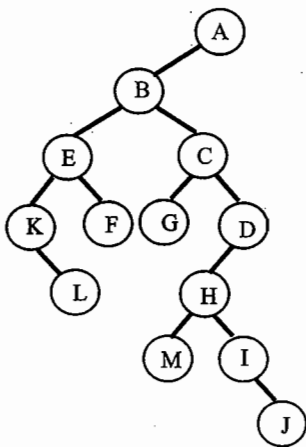
步骤 2: 链接兄弟节点



步骤 3: 打断与右子节点的链接



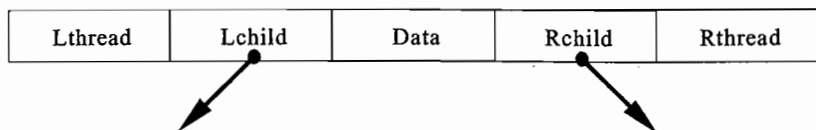
步骤 4: 将兄弟节点顺时针转 45°



## 7.12 引线二叉树

从前面的章节中可看出,有关二叉树链表结构中,许多节点指针指向 NULL 的情形很多,特别是叶节点(leaf node),其左、右指针均指向 NULL。而本节所要介绍的引线二叉树(Threaded binary tree)则是充份应用这些空指针建立引线,使二叉树中的节点能有更紧密的连结,并且能够加快二叉树的遍历速度。

为了区分指针是指向子节点还是为一引线,引线二叉树的节点结构分别对左指针和右指针各增加一个字段来标示指针的类型。其结构如下:



其中 Lthread 和 Rthread 是分别用来指示 Lchild 和 Rchild 是否为指针或引线, 若 child 字段为引线, 则 thread 字段值为 1, 若 child 字段为指针, 则 thread 字段值为 0。节点结构的声明方式如下:

```

struct thread_tree
{
 int Lthread;
 struct thread_tree *left;
 int data;
 struct thread_tree *right;
 int Rthread;
};
typedef struct thread_tree treenode;
typedef treenode *t_btreenode;

```

了解引线二叉树的节点结构及声明方法后, 则要进一步说明引线二叉树的建立方式。事实上, 引线二叉树的建立方式和一般二叉树相同, 只是要额外考虑指针字段和引线字段的内容, 规则如下:

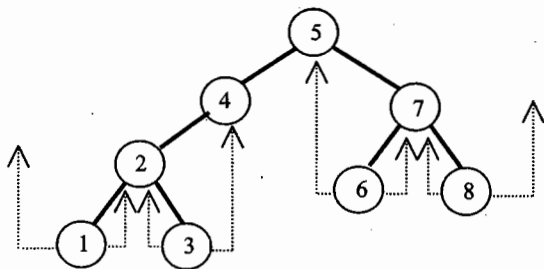
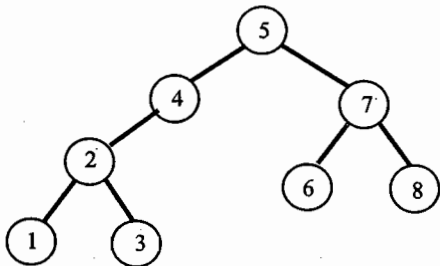
```

if 左指针为空 then
 Lchild 指到在该树中序遍历的前一个节点
 将 Lthread 设为 1
if 右指针为空 then
 Rchild 指到在该树中序遍历的后一个节点
 将 Rthread 设为 1

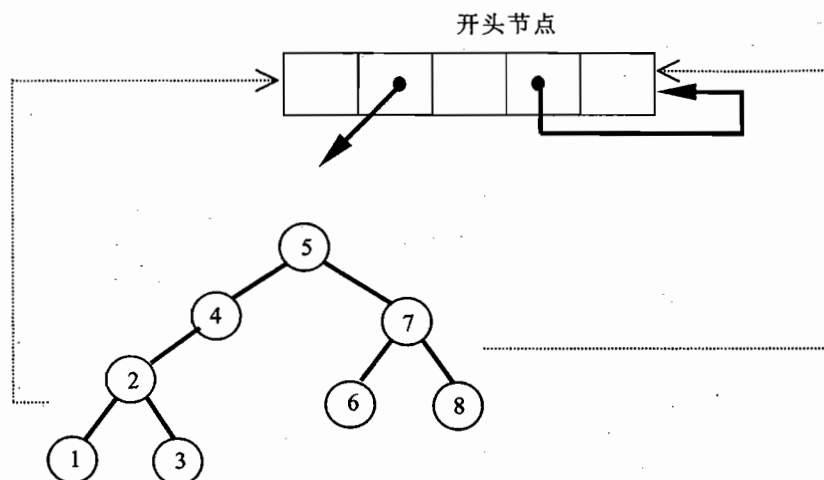
```

如有一棵二叉树如右:

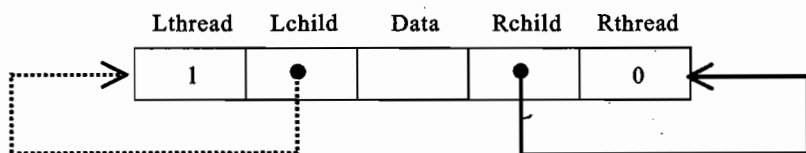
图中二叉数之中序遍历之顺序为: 1 2 3 4 5 6 7 8, 为遵守引线二叉树的建立规则, 以节点 6 为例, 其左指针为空, 则建立左引线指向中序遍历的前一个节点 5, 且其右指针也为空, 则建立右引线指向中序遍历的后一个节点 7。其余的节点依此类推, 则得到结果如下图:



从上图中可看出, 二叉树中序遍历的第一个节点 1 之左引线和最后一个节点 8 之右引线没有连通的节点。为解决这样的问题, 需要再加上一个开头节点, 使原二叉树成为其左子树, 而其右指针则指向开头节点本身。如此一来节点 1 之左引线和节点 8 之右引线均可指向开头节点, 如下图所示:

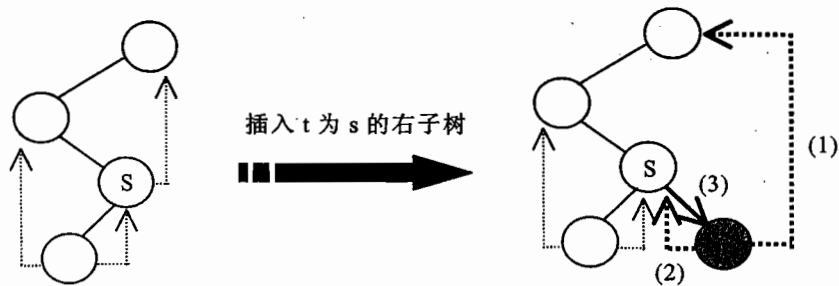


当然，若二叉树为空时，开头节点之左指针会指向 NULL，故会建立左引线指向自己，得到结果如下图：



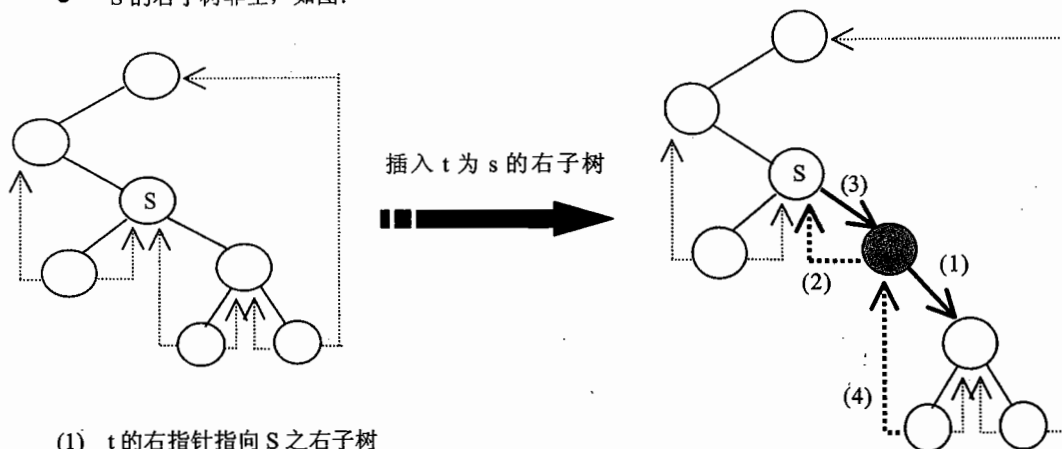
接着要进一步说明如何将一节点插入到引线二叉树中。假设一节点是引线二叉树中的任一节点，欲插入一新节点  $t$  成为节点  $S$  的右子树，需考虑两种状况：

- $S$  的右子树为空，如图：



- (1)  $t$  的右引线指向 root
- (2)  $t$  的左引线指向  $s$
- (3)  $s$  的右指针指向  $t$

- S 的右子树非空, 如图:



- (1) t 的右指针指向 S 之右子树
- (2) t 的左引线指向 S
- (3) S 的右指针指向 t
- (4) S 的原右子树最左边节点的左引线指向 t

若欲插入一新节点 t 成为节点 S 的左子树的方法和插入为右子树的处理方式很类似, 留给读者自行练习。

#### 程序实例:

使用数组来建立引线二叉树, 并以中序遍历的方式将引线二叉树之节点内容打输出来。

#### 程序构思:

数组 `nodelist` 中为建立引线二叉树之节点数据, 利用函数 `insert_node()`——将节点插入引线二叉树中, 最后以中序遍历的方式将节点内容输出来。

#### 程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_13.c */
03 /* 程序目的: 建立引线二叉树 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 struct thread_tree
08 {
09 int data; /*节点数据*/
10 struct thread_tree *Lchild; /*左指针*/
11 struct thread_tree *Rchild; /*右指针*/
12 int Lthread; /*标示是否为左引线*/
13 int Rthread; /*标示是否为右引线*/
14 };
15 typedef struct thread_tree treenode; /*定义新类型树状结构*/
16 typedef treenode *t_btree; /*声明树节点的指针类型*/
17
18 /*-----*/
19 /* 插入引线二叉树的节点 */
20 /*-----*/
21 t_btree insert_node(t_btree root, int node)

```

```

22 {
23 t_btree newnode; /*声明树根指针*/
24 t_btree currentnode; /*声明目前节点指针*/
25 t_btree parentnode; /*声明父节点指针*/
26 t_btree prenode; /*声明后接的节点指针*/
27 int position;
28
29 /*建立新节点的内存空间*/
30 newnode=(t_btree) malloc (sizeof(treenode));
31
32 newnode->data =node; /*存入节点内容*/
33 newnode->Rchild=NULL; /*设置右指针初值*/
34 newnode->Lchild=NULL; /*设置左指针初值*/
35 newnode->Rthread=1; /*默认右指针为引线*/
36 newnode->Lthread=1; /*默认左指针为引线*/
37
38 currentnode=root->Rchild; /*目前的根节点指针*/
39
40 if (currentnode==NULL) /*判断是否为树根*/
41 {
42 root->Rchild=newnode; /*开头节点(head node)*/
43 newnode->Lchild=root; /*左指针指向 root*/
44 newnode->Rchild=root; /*右指针指向 root*/
45 return root; /*返回根指针*/
46 }
47
48 parentnode=root; /*设置父节点为开头节点*/
49 position=0; /*设置前进方向*/
50 while (currentnode!=NULL)
51 {
52 if (currentnode->data > node) /*比较节点的数据值*/
53 {
54 if (position!=-1)
55 {
56 position=-1; /*新的前进方向*/
57 prenode=parentnode; /*记录后接的节点指针*/
58 }
59 parentnode=currentnode; /*保留父节点指针*/
60 if (currentnode->Lthread ==0) /*判断左指针是否为引线*/
61 currentnode=currentnode->Lchild;
62 else
63 currentnode=NULL;
64 }
65 else
66 {
67 if (position!=-1)
68 {
69 position=-1; /*新的前进方向*/
70 prenode=parentnode; /*记录后接的节点指针*/
71 }
72 parentnode=currentnode; /*保留父节点指针*/
73 if (currentnode->Rthread ==0) /*判断右指针是否为引线*/
74 currentnode=currentnode->Rchild;
75 else
76 currentnode=NULL;
77 }
78 }
79 if (parentnode->data > node) /*连结父节点和子节点*/
80 {
81 parentnode->Lthread=0; /*左指针不是引线*/
82 parentnode->Lchild=newnode; /*该节点为父节点之左子树*/

```

```

83 newnode->Lchild=prenode; /*指针指向 prenode*/
84 newnode->Rchild=parentnode; /*指针指向 parent*/
85 }
86 else
87 {
88 parentnode->Rthread=0; /*右指针不为引线*/
89 parentnode->Rchild=newnode; /*该节点为父节点的右子树*/
90 newnode->Lchild=parentnode; /*指针指向 parent*/
91 newnode->Rchild=prenode; /*指针指向 prenode*/
92 }
93 return root; /*返回根节点指针*/
94 }
95 /*-----*/
96 /* 建立引线二叉树 */
97 /*-----*/
98 t_btree create_t_btree(int *data,int len)
99 {
100 t_btree root=NULL; /*根节点指针*/
101 int i;
102
103 /*建立引线二叉树之开头节点之内存空间*/
104 root=(t_btree) malloc (sizeof(treenode));
105
106 root->Rchild=NULL; /*设置开头节点右指针初值*/
107 root->Lchild=root; /*设置开头节点左指针初值*/
108 root->Rthread=0; /*设置开头节点右子树为指针 0*/
109 root->Lthread=1; /*设置开头节点左子树为引线 1*/
110
111 for (i=0; i < len; i++) /*建立树状结构*/
112 root=insert_node(root,data[i]);
113 return root;
114 }
115 /*-----*/
116 /* 引线二叉树中序遍历 */
117 /*-----*/
118 void inorder_print(t_btree root)
119 {
120 t_btree point;
121
122 point=root; /*指向开头指针开始遍历*/
123 do
124 {
125 if (point->Rthread==1) /*右子节点为引线*/
126 point=point->Rchild; /*往右子树前进*/
127 else
128 {
129 point=point->Rchild; /*先到右子节点*/
130 while (point->Lthread !=1) /*左子节点不是引线*/
131 point=point->Lchild; /*往左子节点前进*/
132 }
133 if (point!=root)
134 printf("[%d]\n",point->data); /*输出节点数据*/
135 }while (point!=root); /*找到开头节点*/
136 }
137 /*-----*/
138 /*主程序:建立引线二叉树并以中序遍历打印 */
139 /*-----*/
140 void main()
141 {
142 t_btree root; /*根节点指针*/

```

```

144 int nodelist[10]={5,4,7,2,6,8,1,3}; /*节点数据*/
145 int i;
146 pointer RList,LList,BList,temp;
147
148 /*建立引线二叉树*/
149 root=create_t_btree(nodelist,8);
150 printf("\nThread binary tree node content(inorder):\n");
151 inorder_print(root);
152 }

```

运行结果:

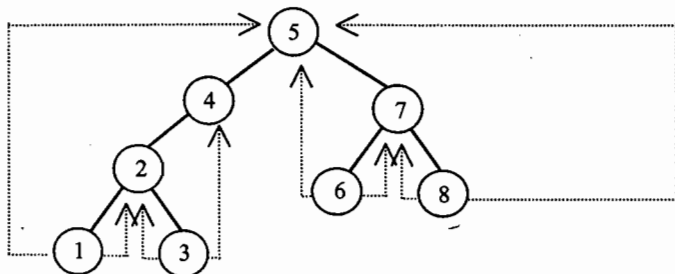
```

C:\DS>Tree_13
Thread binary tree node content(inorder):
[1] [2] [3] [4] [5] [6] [7] [8]

C:\DS>

```

运行程序所建立的引线二叉树如下:



## 7.13 二叉树的应用(表达式)

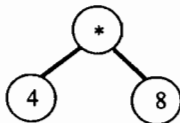
在二叉树的应用当中,最常见的就是表达式的处理。由于表达式的处理需考虑计算的优先级,而二叉树的左右子树也有顺序之分,故若能将表达式数据整理成二叉查找树,则可用来方便处理表达式。

若考虑中序表达式:

$$4 * 8 + 6 / 2$$

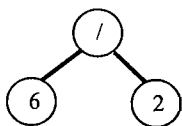
将运算值当做叶节点,运算符号当做非终端节点。

(1) 考虑运算符号“\*”可得:

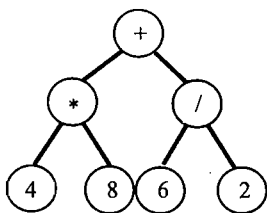




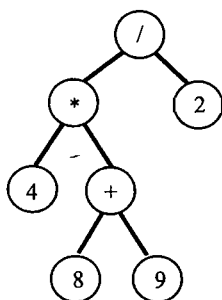
(2) 考虑运算符“/”可得:



(3) 最后处理运算符“+”可得下列完整的运算二叉树:

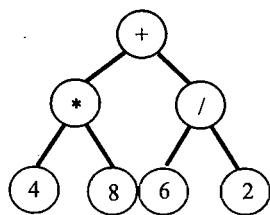


若没有考虑到处理的优先级, 可能产生错误的二叉树, 如下图:



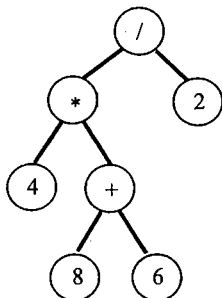
从二叉树计算表达式是从叶节点依序往上层计算, 考虑两者之运算结果及其前序、中序与后序表达式如下:

原二叉树:



preorder: + \* 4 8 / 6 2  
 inorder: 4 \* 8 + 6 / 2  
 postorder: 4 8 \* 6 2 / +  
 运算结果: 35

错误二叉树:



preorder: / \* 4 + 8 6 2  
 inorder: 4 \* 8 + 6 / 2  
 postorder: 4 8 \* 6 2 / +  
 运算结果: 28

可明显地看出两者的结果不同, 故只有二叉树建立正确的表达式方可计算出正确的表达式结果。

## 程序实例:

将表达式以二叉树方式存入数组,以递归方式建立表达式之二叉树状结构,再分别输出前序、中序及后序遍历结果,并计算出表达式之结果。

## 程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Tree_14.c */
03 /* 程序目的: 建立表达式二叉树并计算其结果 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07
08 struct tree /*声明树的结构*/
09 { struct tree *left; /*存放左子树的指针*/
10 char data; /*存放节点数据内容*/
11 struct tree *right; /*存放右子树的指针*/
12 };
13 typedef struct tree treeNode; /*声明新类型树结构*/
14 typedef treeNode *b_tree; /*声明二叉树的链表*/
15
16 /*-----*/
17 /* 使用递归建立二叉树 */
18 /*-----*/
19 b_tree create_btree(int *nodelist,int position)
20 {
21 b_tree newnode; /*声明新节点指针*/
22
23 if (nodelist[position] == 0 || position > 7) /*递归的终止条件*/
24 return NULL;
25 else
26 {
27 /*-----建立新节点的内存空间-----*/
28 newnode=(b_tree) malloc (sizeof(treeNode));
29
30 /*-----建立节点内容-----*/
31 newnode->data=nodelist[position];
32
33 /*-----递归建立左子树-----*/
34 newnode->left=create_btree(nodelist,2*position);
35
36 /*-----递归建立右子树-----*/
37 newnode->right=create_btree(nodelist,2*position+1);
38
39 return newnode;
40 }
41 }
42 /*-----*/
43 /* 二叉树前序遍历打印节点内容 */
44 /*-----*/
45 void preorder(b_tree point)
46 {
47 if (point!=NULL) /*遍历的终止条件*/
48 {
49 printf("%c ",point->data); /*处理打印节点内容*/
50 preorder(point->left); /*处理左子树*/
51 preorder(point->right); /*处理右子树*/
52 }

```

```

53 }
54 /*-----*/
55 /*二叉树中序遍历打印节点内容 */
56 /*-----*/
57 void inorder(b_tree point)
58 {
59 if (point!=NULL) /*遍历的终止条件*/
60 {
61 inorder(point->left); /*处理左子树*/
62 printf("%c ",point->data); /*处理打印节点内容*/
63 inorder(point->right); /*处理右子树*/
64 }
65 }
66 /*-----*/
67 /*二叉树后序遍历打印节点内容 */
68 /*-----*/
69 void postorder(b_tree point)
70 {
71 if (point!=NULL) /*遍历的终止条件*/
72 {
73 postorder(point->left); /*处理左子树*/
74 postorder(point->right); /*处理右子树*/
75 printf("%c ",point->data); /*处理打印节点内容*/
76 }
77 }
78 /*-----*/
79 /* 计算表达式结果值 */
80 /*-----*/
81 int calculate(b_tree point)
82 {
83 int oper1=0; /*前操作数变量*/
84 int oper2=0; /*后操作数变量*/
85
86 if (point->left == NULL && point->right == NULL)
87 return point->data.48;
88 else
89 {
90 oper1=calculate(point->left); /*左子树*/
91 oper2=calculate(point->right); /*右子树*/
92 return get_value(point->data,oper1,oper2);
93 }
94 }
95 /*-----*/
96 /* 抽取运算值并计算 */
97 /*-----*/
98 int get_value(int oper,int oper1,int oper2)
99 {
100 switch ((char) oper)
101 {
102 case '*' : return (oper1*oper2);
103 case '/' : return (oper1/oper2);
104 case '+' : return (oper1+oper2);
105 case '-' : return (oper1.oper2);
106 }
107 }
108 /*-----*/
109 /* 主程序:建立表达式二叉树,并计算结果 */
110 /*-----*/
111 void main()
112 {
113 b_tree root=NULL; /*声明表达式二叉树指针*/

```

```

114 int cal_result; /*计算结果*/
115
116 /*-----声明二叉树数组节点数据-----*/
117 int nodelist[8]={' ', '+', '*', '/', '4', '8', '6', '2' };
118
119 root=create_btree(nodelist,1); /*建立表达式二叉树*/
120 /*-----前序打印-----*/
121 printf("\nPreorder expression: [");
122 preorder(root);
123 printf("]\n");
124 /*-----中序打印-----*/
125 printf("\nInorder expression: [");
126 inorder(root);
127 printf("]\n");
128 /*-----后序打印-----*/
129 printf("\nPostorder expression: [");
130 postorder(root);
131 printf("]\n");
132
133 /*-----计算表达式结果-----*/
134 cal_result=calculate(root);
135 printf("\nCalculate result is [%2d]\n",cal_result);
136 }

```

运行结果:

```

C:\DS>Tree_14
Preorder expression: [+ * 4 8 / 6 2]
Inorder expression: [4 * 8 + 6 / 2]
Postorder expression: [4 8 * 6 2 / +]

Calculate result is [35]

C:\DS>

```

## 【习题】

### 一、复习:

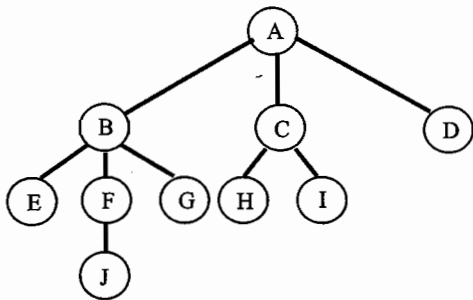
1. 树可为空,但二叉树不可为空。
2. 二叉树的分支度必为 0、1 或 2,而树的分支度则没有限制。
3. 一般树和二叉树的子树都有顺序关系。
4. 若一棵二叉树为满二叉树,则也必为完全二叉树。
5. 若一棵二叉树为完全二叉树,则也必为满二叉树。
6. 请问一深度为  $k$  之满二叉树有几个节点?  
(a)  $2^k$     (b)  $2^k-1$     (c)  $2^{k-1}$     (d)  $2^{k+1}$
7. 在二叉树中阶层为  $i$  之节点最多有几个?  
(a)  $2^i$     (b)  $2^i-1$     (c)  $2^{i-1}$     (d)  $2^{i+1}$
8. 对于任一个非空二叉树而言,若分支度为  $i$  之节点各有  $n_i$  个,则下列哪一个正确?  
(a)  $n_1 = n_2 + 1$     (b)  $n_2 = n_1 + 1$     (c)  $n_2 = n_0 + 1$     (d)  $n_0 = n_2 + 1$
9. 请问下列何种表示法用于歪斜树时会造成内存使用率偏低  
(a) 链表表示法    (b) 数组表示法    (c) 结构数组表示法

10. 试解释下列名词:

- (1) 树
- (2) 二叉树
- (3) 满二叉树
- (4) 完全二叉树
- (5) 歪斜树
- (6) 引线二叉树

二、应用:

1. 试比较树和二叉树。
2. 试分别写出下列二叉树之前序、中序及后序遍历之顺序。
3. 请按照下列节点的输入顺序建构一引线二叉树, 需明确表示出各节点之指针及引线。  
节点输入顺序为: 4 8 2 3 6 5 9 7 1
4. 试绘出下列表达式之二叉树结构
  - (1) 前序:  $+*/AB-CDE$
  - (2) 中序:  $Y=(A+B/C-D)*(E*(F+G))$
5. 试将下面的树转换成二叉树。



6. 试写出一“复制二叉树”的函数。
7. 试写出一“列出二叉树中所有叶节点”的函数。
8. 试写出一“计算二叉树所有节点个数”的函数。
9. 试写出一递归函数找出二叉树中最长路径的长度。
10. 试写出一“映像二叉树”的函数。

# 排 序

## 第 8 章

- ◆ 何谓排序
- ◆ 内部排序法——交换式排序
- ◆ 内部排序法——选择式排序
- ◆ 内部排序法——插入式排序
- ◆ 外部排序法——合并排序法
- ◆ 排序法的效率比较

## 8.1 何谓排序

### 8.1.1 排序的意义

所谓排序(Sort)是将一群数据,依指定顺序所进行的排列过程。最常见的有“由小到大”的“递增顺序”和“由大到小”的“递减顺序”。

以下以数组的结构作说明:

排序前:

|      | 0  | 1  | 2 | 3  | 4 | 5  | 6  |
|------|----|----|---|----|---|----|----|
| data | 27 | 16 | 9 | 31 | 7 | 24 | 17 |

排序后:

(由小到大)

|      | 0 | 1 | 2  | 3  | 4  | 5  | 6  |
|------|---|---|----|----|----|----|----|
| data | 7 | 9 | 16 | 17 | 24 | 27 | 31 |

(由大到小)

|      | 0  | 1  | 2  | 3  | 4  | 5 | 6 |
|------|----|----|----|----|----|---|---|
| Data | 31 | 27 | 24 | 17 | 16 | 9 | 7 |

### 8.1.2 排序的特性——稳定性与不稳定性

所谓“稳定性排序”即是排序过后能使值相同的数据,保持原顺序中之相对位置。反之,则称为不稳定性排序。

例如:

|      | 0  | 1  | 2    | 3  | 4 | 5  | 6    | 7  |
|------|----|----|------|----|---|----|------|----|
| Data | 27 | 16 | 9(1) | 31 | 7 | 24 | 9(2) | 17 |

9(1)和 9(2)为值相同的数据

顺序: 9(1)在 9(2)之前

稳定性排序的结果:

|      | 0 | 1    | 2    | 3  | 4  | 5  | 6  | 7  |
|------|---|------|------|----|----|----|----|----|
| data | 7 | 9(1) | 9(2) | 16 | 17 | 24 | 27 | 31 |

顺序: 9(1)在 9(2)之前 → 维持原来的相对位置

不稳定性排序的结果:

|      |   |      |      |    |    |    |    |    |
|------|---|------|------|----|----|----|----|----|
|      | 0 | 1    | 2    | 3  | 4  | 5  | 6  | 7  |
| data | 7 | 9(2) | 9(1) | 16 | 17 | 24 | 27 | 31 |

顺序: 9(2)在 9(1)之前 → 和原来的相对位置不同

### 8.1.3 排序的分类

排序的分类大致上可分为两种:

(1) 内部排序 (Internal Sort)

将欲处理的数据整个存放到内部存储器中做排序, 数据可被随机存取。

内部排序法依排序的方式可分为 3 种:

(a) 交换式排序法

(b) 选择式排序法

(c) 插入式排序法

(2) 外部排序 (External Sort)

欲处理的数据量过于庞大, 无法全部存放到内部存储器, 必须借助外部辅助内存 (比如: 磁盘, 磁带), 由于数据是存在外部内存中, 故数据不可随机被存取。

外部排序又分为两种:

(a) 合并排序法

(b) 直接合并排序法

以上各种内部排序法及外部排序法将在本章各小节中, 以递增的排序原则为读者做详细的说明。

## 8.2 内部排序法——交换式排序

内部排序法中的交换式排序, 是运用数据值比较后, 依判断规则对数据位置进行交换, 以达到排序的目的。

交换式排序法又可分为两种:

(1) 冒泡排序法 (Bubble Sort)

(2) 快速排序法 (Quick Sort)

### 8.2.1 冒泡排序法

方法:

将相邻的两个数据加以比较, 若左边的值大于右边的值, 则将此两个值互相交换; 若左边的值小于等于右边的值, 则此两个值的位置不变。右边的值继续和下一个值做比较, 重复此动作, 直到比较到最后一个值。

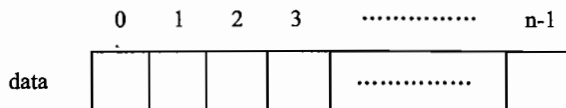
```

if (左边的值 > 右边的值) then
 此两个值的位置互换
else /* 左边的值 ≤ 右边的值 */
 此两个值的位置不变
 右边的值继续和下一个值比较

```



假设共有  $n$  个数据  $\text{data}[0] \sim \text{data}[n-1]$ :



第 1 回 (从  $\text{data}[0]$  到  $\text{data}[n-1]$ ): 最大者存放在  $\text{data}[n-1]$

第 2 回 (从  $\text{data}[0]$  到  $\text{data}[n-2]$ ): 本回最大者存放在  $\text{data}[n-2]$

...

...

第  $(n-1)$  回 ( $\text{data}[0]$  和  $\text{data}[1]$ ) : 最小者存放在  $\text{data}[0]$

此方法在每比较一回就会以交换位置的方式将该回的最大者移向数组的尾端, 就像气泡从水底浮向水面一样, 到水面时气泡最大, 故称为冒泡排序法。

举例说明:

|      |    |    |   |    |
|------|----|----|---|----|
|      | 0  | 1  | 2 | 3  |
| data | 37 | 96 | 8 | 54 |

$\text{data}[0] \sim \text{data}[3]$  共 4 个数据, 共需比 3 回 (6 次)

◆ 第 1 回 (从  $\text{data}[0]$  到  $\text{data}[3]$ ---4 个)  $\rightarrow$  比 3 次

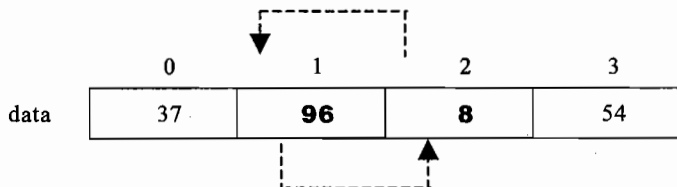
(1) 比较  $\text{data}[0]$  &  $\text{data}[1]$

$37 < 96 \rightarrow$  不交换

|      |           |           |   |    |
|------|-----------|-----------|---|----|
|      | 0         | 1         | 2 | 3  |
| data | <b>37</b> | <b>96</b> | 8 | 54 |

(2) 比较  $\text{data}[1]$  &  $\text{data}[2]$

$96 > 8 \rightarrow$  交换

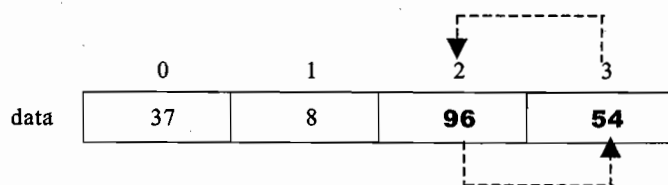


交换结果:

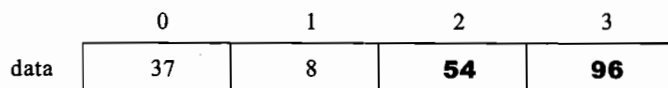
|      |    |          |           |    |
|------|----|----------|-----------|----|
|      | 0  | 1        | 2         | 3  |
| data | 37 | <b>8</b> | <b>96</b> | 54 |

(3) 比较  $\text{data}[2]$  &  $\text{data}[3]$

$96 > 54 \rightarrow$  交换



交换结果:

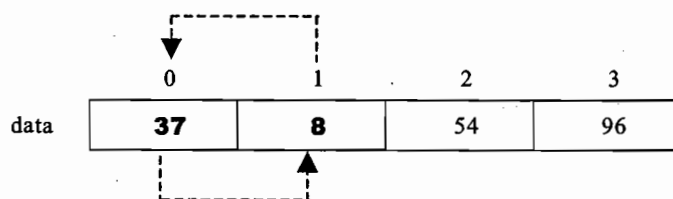


※第1回: 最大者 96 存放在最右端 data[3]

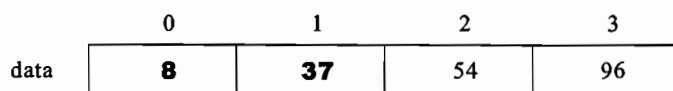
◆ 第2回(从 data[0]比到 data[2]---3个)→比2次

(1) 比较 data[0] & data[1]

$37 > 8 \rightarrow$  交换

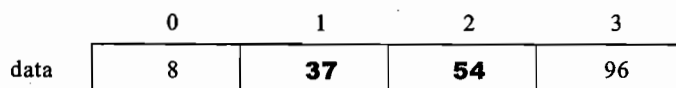


交换结果:



(2) 比较 data[1] & data[2]

$37 < 54 \rightarrow$  不交换

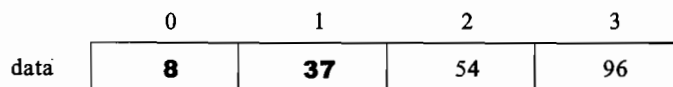


※第2回: 本回最大者 54 存放在 data[2]

◆ 第3回(从 data[0]比到 data[1]---2个)→比1次

(1) 比较 data[0] & data[1]

$8 < 37 \rightarrow$  不交换



※第3回: 大者 37 存放在 data[1]; 最小者 8 存放在 data[0]

冒泡排序法的优点是, 若数据已有部分排好序, 则使用冒泡排序法可以很快速地完成排序。其缺点

是会反复扫描数据，比较相邻的两个数据，速度不快也没有效率。它是属于稳定性排序法，故相同的数值在排序过后其顺序和排序前的顺序是一样的。

- 最佳状况：欲排序数据的顺序恰与排序后的顺序相同

例如： 排序前 --- 1 2 3 4 5

排序后 --- 1 2 3 4 5

- 最坏状况：排序前的顺序和所要的排列结果恰好完全相反

例如： 排序前 --- 5 4 3 2 1

排序后 --- 1 2 3 4 5

另外冒泡排序法的空间复杂度为  $O(1)$ ，因为只需要一个额外的空间来存储交换时的暂存数值。而时间复杂度的平均状况为  $O(n^2)$ ，最坏状况为  $O(n^2)$ ，因为可能每次都要查找到最后。但如果是已经排好序的元素列，则时间复杂度为最佳状况  $O(n)$ 。

#### 程序实例：

使用冒泡排序法设计一个排序程序。

#### 程序构思：

1. 读入欲排序的数值
2. 使用冒泡排序法(两两比较，大者放后面)，只要有交换的动作即打印目前排序的结果
3. 打印最终排序结果

#### 程序源代码：

```
01 /* ===== Program Description ===== */
02 /* 程序名称: Sort_01.c */
03 /* 程序目的: 使用冒泡排序法设计一个排序程序 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 冒泡排序法的比较与交换 */
08 /*-----*/
09 void bubbleSort(int *list, int index)
10 { int i,j,
11 int change; /*记录数值是否有交换位置*/
12 int temp; /*于数值交换时的暂存变量*/
13
14
15 while (!change)
16 {
17 change=1;
18 for (j=index; j>0; j--) /*外层循环*/
19 {
20 for (i=0; i<j-1; i++) /*内层循环*/
21 {
22 if (list[i] > list[i+1]) /*前者较后者大*/
23 { temp=list[i+1]; /*两数值交换位置*/
24 list[i+1]=list[i];
25 list[i]=temp;
26 change=0;
27 }
28 }
29 /*-----打印目前的排序结果-----*/
30 printf("\n Current sorting result:");
31 for (i=0; i<index; i++)
```

```
31
32 printf("%d ",list[i]);
33 }
34 }
35 }
36 }
37 }
38 }
39 /*-----*/
40 /* 主程序:输入数值数据→进行冒泡排序→打印排序结果 */
41 /*-----*/
42 void main()
43 { int list[20]; /*默认数组最大长度为20*/
44 int i,index, /*数组索引*/
45 int node; /*读入输入值所使用的暂存变量*/
46
47 printf("\n Please input the values you want to sort(Exit for 0):\n");
48 index=0;
49
50 /*-----读取数值存到数组中-----*/
51 scanf("%d", &node); /*读取输入数值并存到变量中*/
52 while (node != 0) /*数列尚未结束*/
53 {
54 list[index]=node; /*将变量中的数值存到数组中*/
55 index=index+1;
56 scanf("%d",&node);
57 }
58
59
60 /*-----进行冒泡排序-----*/
61 bubbleSort(list,index);
62
63 /*-----打印最终的排序结果----- */
64 printf("\n Final sorting result:");
65 for (i=0; i<index; i++)
66 {
67 printf("%d ",list[i]);
68 }
```

运行结果:

```
C:\DS>Sort_01
Please input the values you want to sort(Exit for 0):
32 18 41 23 2 56 36 67 0

Current sorting result:18 32 41 23 2 56 36 67
Current sorting result:18 32 23 41 2 56 36 67
Current sorting result:18 23 32 41 2 56 36 67
Current sorting result:18 23 32 2 41 56 36 67
Current sorting result:18 23 2 32 41 56 36 67
Current sorting result:18 2 23 32 41 56 36 67
Current sorting result:2 18 23 32 41 56 36 67
Current sorting result:2 18 23 32 41 36 56 67
Current sorting result:2 18 23 32 36 41 56 67

Final sorting result: 2 18 23 32 36 41 56 67
C:\DS>
```

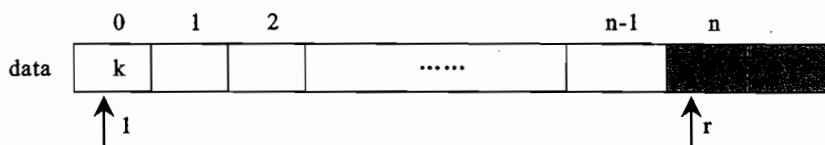
## 8.2.2 快速排序法

方法:

假设有  $n$  个数据  $data[0] \sim data[n-1]$

设立指针  $l = data[0] = k$ ; 指向第 1 个位置  $data[0]$ ,  $i = 0$

指针  $r = data[n-1]$ ; 指向第  $n$  个位置  $data[n]$ ,  $j = n$



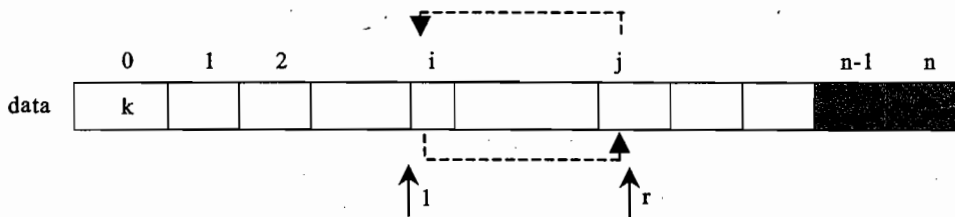
步骤 1:  $l$  往右找, 直到找到比  $l$  值大时停止, 假设停止于  $data[i]$

$r$  往左找, 直到找到比  $r$  值小时停止, 假设停止于  $data[j]$

可能有两种状况:

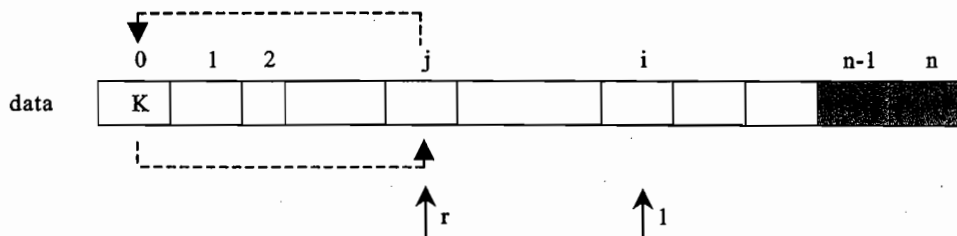
if ( $i < j$ ) then

$data[i]$  和  $data[j]$  的内容值交换

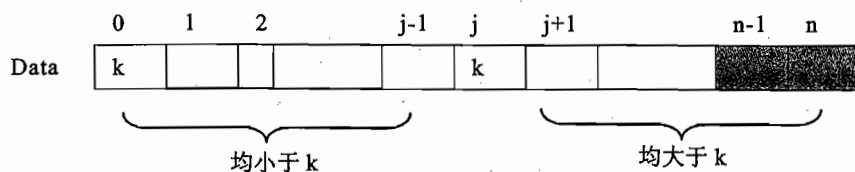


if ( $i \geq j$ ) then

此数组的第一个值和  $data[j]$  的内容值交换



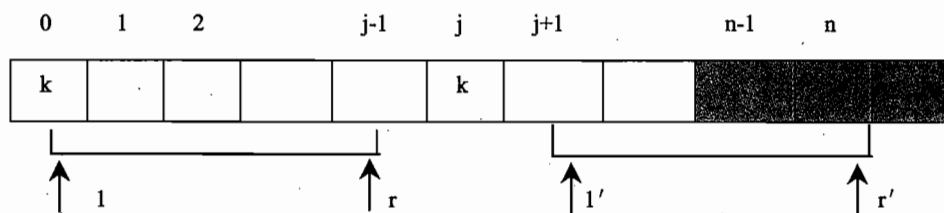
交换后  $k$  值已找到其所属的位置, 并将所有数据切割成两部分, 其左边数据均小于  $k$ , 其右边数据均大于  $k$ 。



步骤2: 每次被分割的区块, 再分别设立  $l$  及  $r$  指针,

$l$  = 区块第 1 个值, 指向区块第 1 个位置

$r$  = 区块最后 1 个值, 指向 (最后 + 1) 的位置



各个区块再个别进行步骤 1, 直到所有区块均已排序完成。

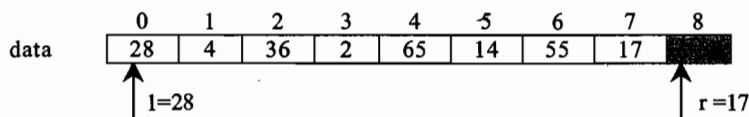
举例说明:

|      |    |   |    |   |    |    |    |    |
|------|----|---|----|---|----|----|----|----|
|      | 0  | 1 | 2  | 3 | 4  | 5  | 6  | 7  |
| Data | 28 | 4 | 36 | 2 | 65 | 14 | 55 | 17 |

$\text{data}[0] \sim \text{data}[7]$  共 8 个数据

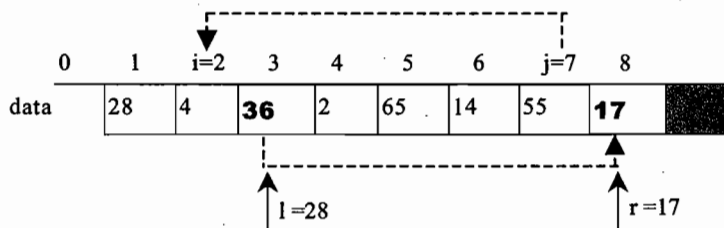
(1)  $l = \text{data}[0] = 28$ , 指向  $\text{data}[0]$ ,  $i = 0$

$r = \text{data}[n-1] = \text{data}[7] = 17$ , 指向  $\text{data}[8]$ ,  $j = 8$



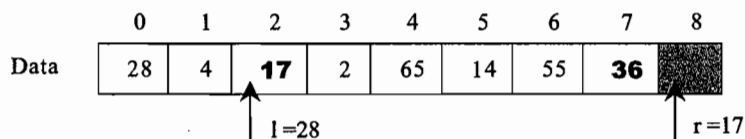
(2)  $l$  往右找, 直到找到  $\geq 28$  时停止,  $i = 2$

$r$  往左找, 直到找到  $\leq 17$  时停止,  $j = 7$

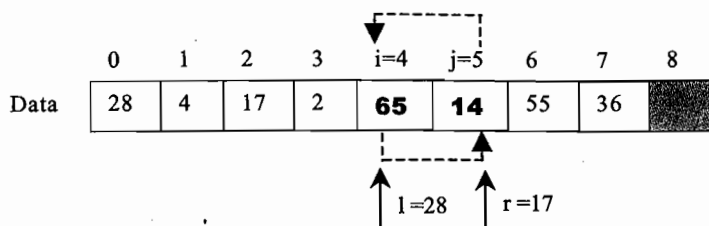


$i < j$  ( $2 < 7$ ) 故  $\text{data}[2]$  和  $\text{data}[7]$  交换

(3)

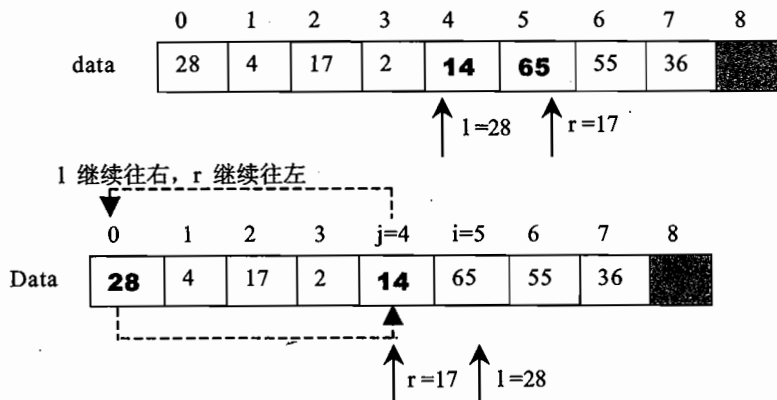


$l$  继续往右,  $r$  继续往左



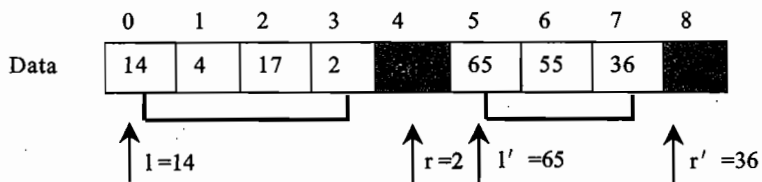
$i < j$  ( $4 < 5$ ) 故 data[4]和 data[5]交换

(4)



$i > j$  ( $5 > 4$ ) 故 data[0]和 data[4]交换

(5)



“28”已找到其所属的位置, 并将数组切割成两部分

※data[0] ~ data[3]:  $l = 14$ , 指向 data[0]

$r = 2$ , 指向 data[4]

※data[5] ~ data[7]:  $l = 65$ , 指向 data[5]

$r = 36$ , 指向 data[8]

此两区块再分别进行相同的排序动作, 直到所有被切割的区块均排好序, 则数据排序完成。

以表格的方式来表示排序的过程, 中括号所包含部分, 为尚待排序的区块。

|     | $R_0$ | $R_1$ | $R_2$ | $R_3$  | $R_4$ | $R_5$ | $R_6$ | $R_7$ |
|-----|-------|-------|-------|--------|-------|-------|-------|-------|
| (1) | [ 28  | 4     | 36    | 2      | 65    | 14    | 55    | 17 ]  |
| (2) | [ 14  | 4     | 17    | 2 ]    | 28    | [ 65  | 55    | 36 ]  |
| (3) | [ 2   | 4 ]   | 14    | [ 17 ] | 28    | [ 65  | 55    | 36 ]  |
| (4) | 2     | 4     | 14    | [ 17 ] | 28    | [ 65  | 55    | 36 ]  |
| (5) | 2     | 4     | 14    | 17     | 28    | [ 65  | 55    | 36 ]  |
| (6) | 2     | 4     | 14    | 17     | 28    | 36    | [ 55  | 65 ]  |
| (7) | 2     | 4     | 14    | 17     | 28    | 36    | 55    | 65    |

快速排序法的特性是不稳定性排序,即相同的数值在排序过后其顺序和排序前的顺序不一样。当每次分割的两个区块都均匀小时为最佳状况。相反的,每次分割的两个区块大小相差很多(例如一半为0,另一半为  $n-1$ )时,则为最坏状况。

另外空间复杂度即是使用递归的深度,平均状况的空间复杂度介于  $O(\log_2 n)$  和  $O(n)$  之间,若是最佳状况则为  $O(\log_2 n)$ ,最坏状况为  $O(n)$ 。

而平均状况和最佳状况的时间复杂度均为  $O(n * \log_2 n)$ ,最坏状况为  $O(n^2)$ 。

程序实例:

使用快速排序法设计一个排序程序。

程序构思:

1. 读入欲排序的数值
2. 使用快速排序法
  - (1) 设置左右端指针(i-左指针,j-右指针)
  - (2) 设分割指针 pivot
  - (3) i 往右找比 pivot 大时停止, j 往左找比 pivot 小时停止
  - (4) 若  $i < j \rightarrow$  list[i] 和 list[j] 内容值对调  
 $i \geq j \rightarrow$  list[left] 和 list[j] 内容值对调
  - (5) pivot 找到其位置,并打输出对调后的排序结果
  - (6) 排序 pivot 左边的元素 QuickSort(左边)
  - (7) 排序 pivot 右边的元素 QuickSort(右边)
3. 打印最终排序结果

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Sort_02.c */
03 /* 程序目的: 使用快速排序法设计一个排序程序 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 快速排序法的递归处理 */
08 /*-----*/
09 void QuickSort(int *list, int left, int right, int index)
10 { int i, j, k,
11 int pivot, /*分割指针*/

```



```

12 int temp; /*于数值交换时的暂存变量*/
13
14 i=left; j=right+1; /*设置 i , j 分别为数组的左右指针*/
15 pivot=list[left]; /*取最左边的元素*/
16
17 if (i < j)
18 { do
19 { /*从左往右找比pivot 大的值*/
20 do
21 { i ++;
22 } while (list[i] <= pivot && i <= right);
23
24 /*从右往左找比pivot 小的值*/
25 do
26 { j--;
27 } while (list[j] >= pivot && j > left);
28
29 if (i < j) /*交换 list[i] . list[j]的值*/
30 {
31 temp=list[i];
32 list[i]=list[j];
33 list[j]=temp;
34 printf("\n *Current sorting result:");
35 for (k=0;k<index;k++)
36 { printf("%d ",list[k]);
37 }
38 }
39 }while (i < j);
40
41 temp=list[left]; /*交换 list[left] . list[j]的值*/
42 list[left]=list[j];
43 list[j]=temp;
44
45 /*-----打印目前的排序结果-----*/ printf("\n
46 #Current sorting result:");
47 for (k=0;k<index;k++)
48 { printf("%d ",list[k]);
49 }
50
51 QuickSort(list , left , j-1,index); /*排序左半边*/
52 QuickSort(list , j+1, right,index); /*排序右半边*/
53 }
54 }
55
56
57 /*-----*/
58 /* 主程序:输入数值数据→进行快速排序→打印排序结果 */
59 /*-----*/
60 void main()
61 { int list[20]; /*设置数组最大长度为 20*/
62 int node; /*读入输入值所使用的暂存变量*/
63 int i,index,
64
65 printf("\n Please input the values you want to sort(Exit for 0):\n");
66
67 index=0;
68
69 /*-----读取数值存到数组中-----*/
70 scanf("%d", &node); /*读输入值存到变量 node*/
71 while (node != 0) /*数列尚未结束*/
72 {

```

```

73 list[index]=node;
74 index=index+1;
75 scanf("%d",&node);
76 }
77
78 /*-----进行快速排序-----*/
79 QuickSort(list,0,index-1,index);
80
81 /*-----打印最终的排序结果-----*/
82 printf("\n Final sorting result:");
83 for (i=0;i<index;i++)
84 {
85 printf("%d ",list[i]);
86 }

```

运行结果:

```

C:\DS>Sort_02
Please input the values you want to sort(Exit for 0):
4 6 2 9 5 1 7 3 0

*Current sorting result:4 3 2 9 5 1 7 6
*Current sorting result:4 3 2 1 5 9 7 6
#Current sorting result:1 3 2 4 5 9 7 6
#Current sorting result:1 3 2 4 5 9 7 6
#Current sorting result:1 2 3 4 5 9 7 6
#Current sorting result:1 2 3 4 5 9 7 6
#Current sorting result:1 2 3 4 5 9 7 6
#Current sorting result:1 2 3 4 5 6 7 9
#Current sorting result:1 2 3 4 5 6 7 9
#Current sorting result:1 2 3 4 5 6 7 9

Final sorting result: 1 2 3 4 5 6 7 9

C:\DS>

```

## 8.3 内部排序法——选择式排序

内部排序法中的选择式排序,是从欲排序的数据中,按指定的规则选出某一元素,经过和其它元素重整,再依原则交换位置后达到排序的目的。

选择式排序又可分为两种:

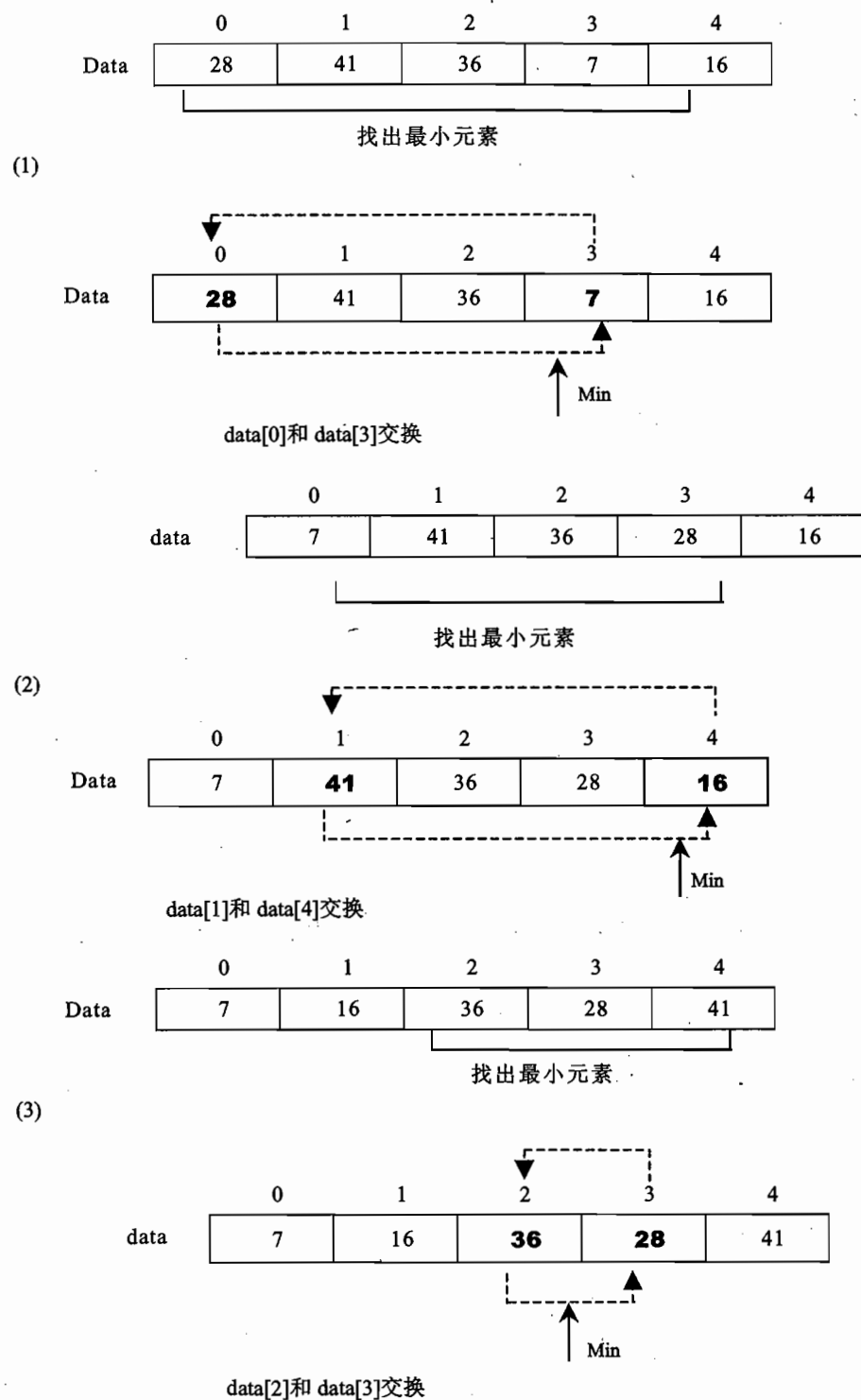
1. 选择排序法 (Selection Sort)
2. 累堆排序法 (Heap Sort)

### 8.3.1 选择排序法

方法:

从欲排序的  $n$  个数据中,以线性查找的方式找出最小的元素和第一个元素交换,再从剩下的  $(n-1)$  个数据中,找出最小的元素和第二个元素交换,以此类推,直到所有元素均已排序完成。

举例说明:



|      |   |    |    |    |    |
|------|---|----|----|----|----|
|      | 0 | 1  | 2  | 3  | 4  |
| Data | 7 | 16 | 28 | 36 | 41 |

(4) 排序完成

|      |   |    |    |    |    |
|------|---|----|----|----|----|
|      | 0 | 1  | 2  | 3  | 4  |
| Data | 7 | 16 | 28 | 36 | 41 |

以优点来看, 选择排序法是最简单的排序法, 但选择排序法所需的排序时间比其它排序法来的长。另外, 如果使用额外的数组来处理时, 空间复杂度为  $O(n)$ , 若只用一个额外的暂存变量来处理数值交换, 则为  $O(1)$ 。因为每次都要查找所有的数值找出最大或最小值, 故时间复杂度均为  $O(n^2)$ 。

程序实例:

使用选择排序法设计一个排序程序。

程序构思:

1. 读入欲排序的数值
2. 使用快速排序法
  - (1) 找数列中最小值与数组第 1 个数值对调, 打印目前排序结果
  - (2) 找数列中次小值与数组第 2 个数值对调, 打印目前排序结果
  - (3) 依此类推直到所有数值排序好
3. 打印最终排序结果

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Sort_03.c */
03 /* 程序目的: 使用选择排序法设计一个排序程序 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 选择排序法的处理 */
08 /*-----*/
09 void SelectSort(int *list, int index)
10 {
11 int i, j, k,
12 int minnode, /*存储最小数值*/
13 int IndexMin, /*存储最小数值的索引值*/
14 int temp; /*于数值交换时的暂存变量*/
15
16 for (i=0 ; i < index -1 ; i++)
17 {
18 minnode=32767; /*目前最小数值*/
19 IndexMin=0; /*存储最小数值的索引值*/
20
21 for (j = i ; j < index ; j++)
22 {
23 if (list[j] < minnode) /*找到最小值*/

```

```

23
24 minnode= list[j]; /*将最小值存到变量 minnode*/
25 IndexMin=j;
26 }
27 temp = list[i]; /*交换两数值*/
28 list[i]=list[IndexMin];
29 list[IndexMin]=temp;
30 }
31
32 /******打印目前排序结果******/
33 printf("\n Current sorting result:");
34 for (k=0;k<index;k++)
35 {
36 printf("%d ",list[k]);
37 }
38 }
39 }
40
41 /*-----*/
42 /***主程序:输入数值数据→进行选择排序→打印排序结果***/
43 /*-----*/
44 void main()
45 { int list[20]; /*设置数组最大长度为 20*/
46 int i,index,
47 int node; /*读入输入值所使用的暂存变量*/
48
49 printf("\n Please input the values you want to sort(Exit for 0):\n");
50
51 index=0;
52
53 /******读取数值存入数组中******/
54 scanf("%d", &node); /*读输入值存到变量 node*/
55 while (node != 0) /*数列尚未结束*/
56 {
57 list[index]=node;
58 index=index+1;
59 scanf("%d",&node);
60 }
61
62 /******进行选择排序******/
63 SelectSort(list, index);
64
65
66 /******打印最终排序结果******/
67 printf("\n Final sorting result:");
68 for (i=0; i<index; i++)
69 {
70 printf("%d ",list[i]);
71 }
72 }

```

运行结果:

```

C:\DS>Sort_03
Please input the values you want to sort(Exit for 0):
34 48 39 21 56 17 80 52 0

Current sorting result :17 48 39 21 56 34 80 52
Current sorting result :17 21 48 39 56 34 80 52
Current sorting result :17 21 34 39 56 48 80 52

```

```
Current sorting result :17 21 34 39 56 48 80 52
Current sorting result :17 21 34 39 48 56 80 52
Current sorting result :17 21 34 39 48 52 80 56
Current sorting result :17 21 34 39 48 52 56 80
```

```
Final sorting result: 17 21 34 39 48 52 56 80
```

```
C:\DS>
```

### 8.3.2 累堆排序法

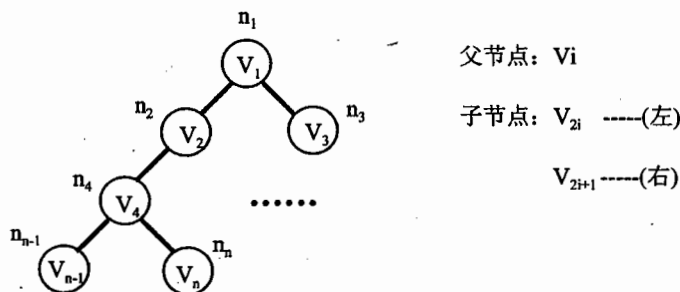
累堆(Heap)的外观为完全二叉树的根最小或最大树,而累堆排序(Heap Sort)所用的堆是“最大堆(即“堆顶元素最大的堆”)”。

方法:

(1) 将欲排序的  $n$  个值存入“数组”

| 0 | 1     | 2     | 3     |       |  | $n-1$     | $n$   |
|---|-------|-------|-------|-------|--|-----------|-------|
|   | $V_1$ | $V_2$ | $V_3$ | ..... |  | $V_{n-1}$ | $V_n$ |

其二叉树表示法为:



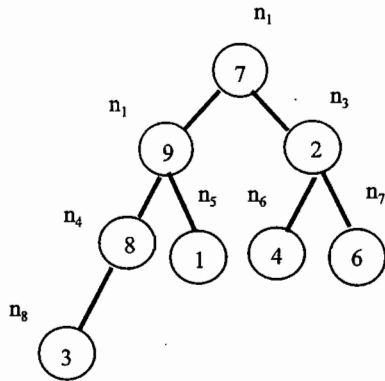
- (2) 将此二叉树创建成最大堆
- (3) 取出树根的值, 将剩下的值再排成堆
- (4) 重复 (3) 的操作, 直到所有值均已输出为止

举例说明:

(1) 将欲排序的 8 个值 “7 9 2 8 1 4 6 3” 存入数组

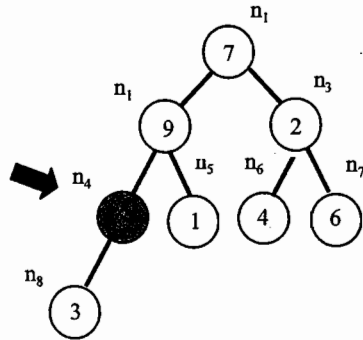
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
|   | 7 | 9 | 2 | 8 | 1 | 4 | 6 | 3 |

其二叉树表示法为:



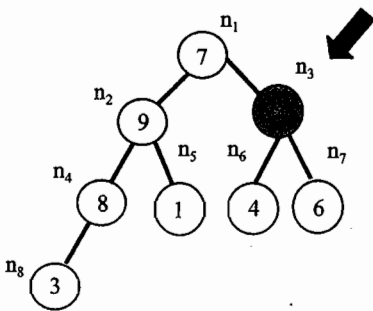
(2) 将此二叉树建立成最大堆( $n=8 \rightarrow 8/2=4 \rightarrow$  开始调整的节点索引值)

<1>考虑  $n_4$

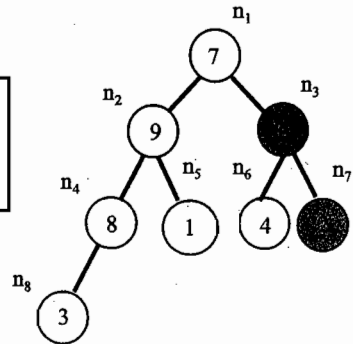


$n_4 > n_8$   
→ 不变

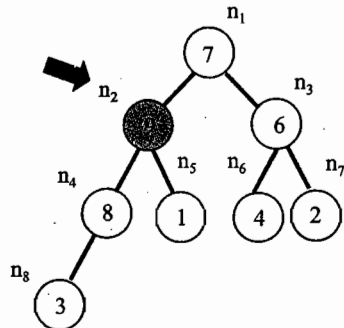
<2>考虑  $n_3$



$n_3 < n_7$   
→ 交换

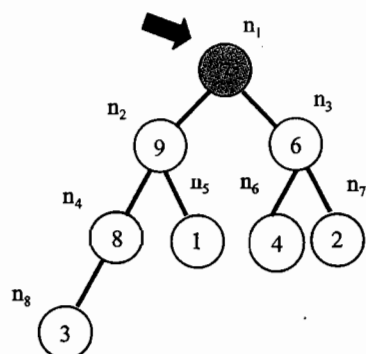


<3>考虑  $n_2$

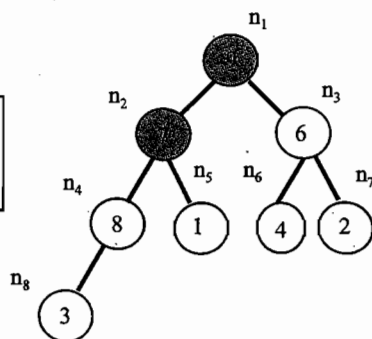


$n_4 < n_2$   
 $n_5 < n_2$   
→ 不变

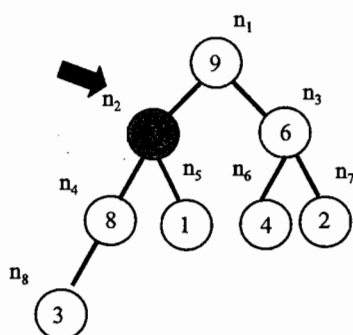
<4>考虑  $n_1$



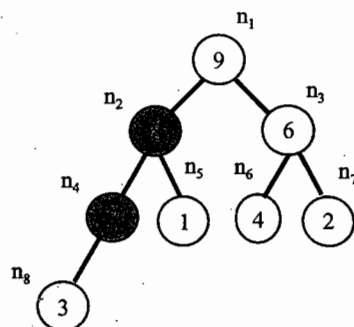
$n_1 < n_2$   
→ 交换



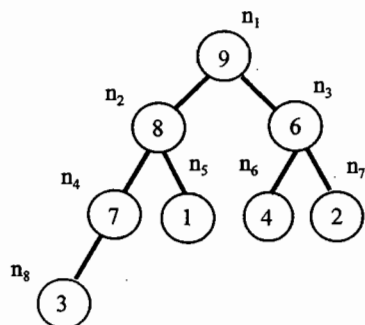
考虑  $n_2$



$n_2 < n_4$   
→ 交换

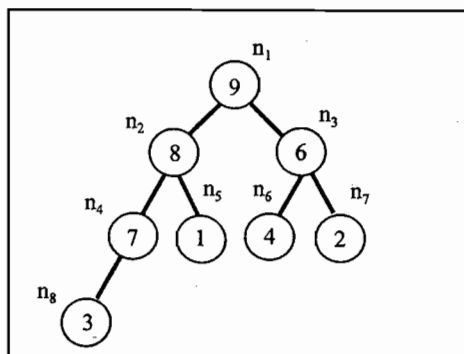


考虑  $n_4$



$n_8 < n_4$   
→ 不变

完成的最大堆



相对应之数组结构排列：

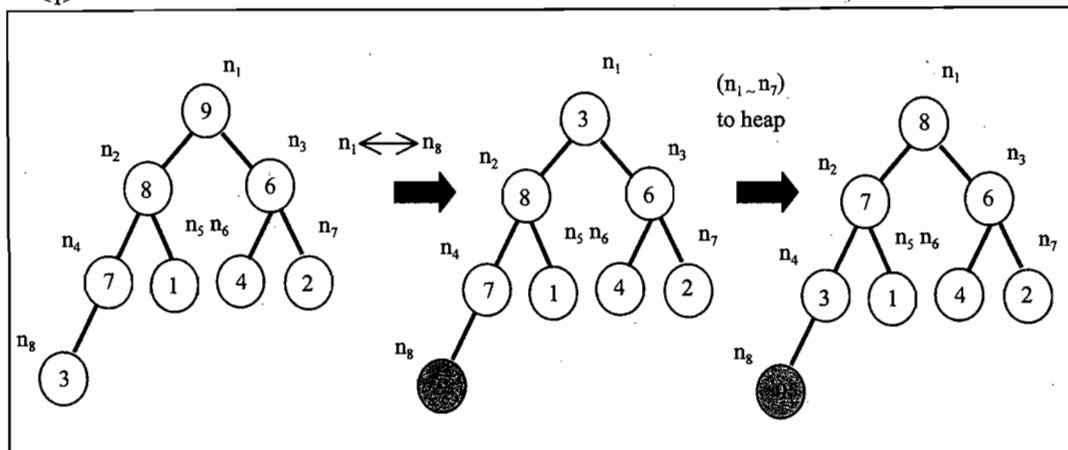
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 9 | 8 | 6 | 7 | 1 | 4 | 2 | 3 |   |



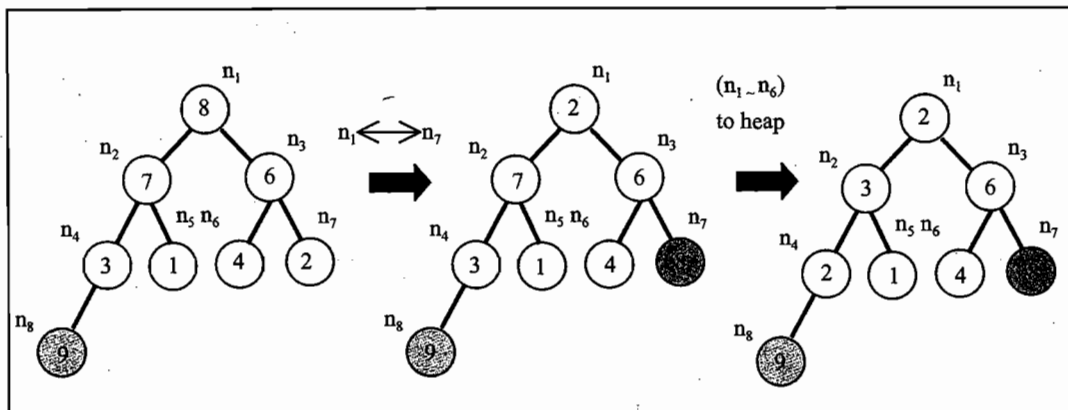
(3) 取出树根的值，将剩下的值再排成堆

※所谓“取出”并非真的是从数组中删除，而是和最后一个节点交换

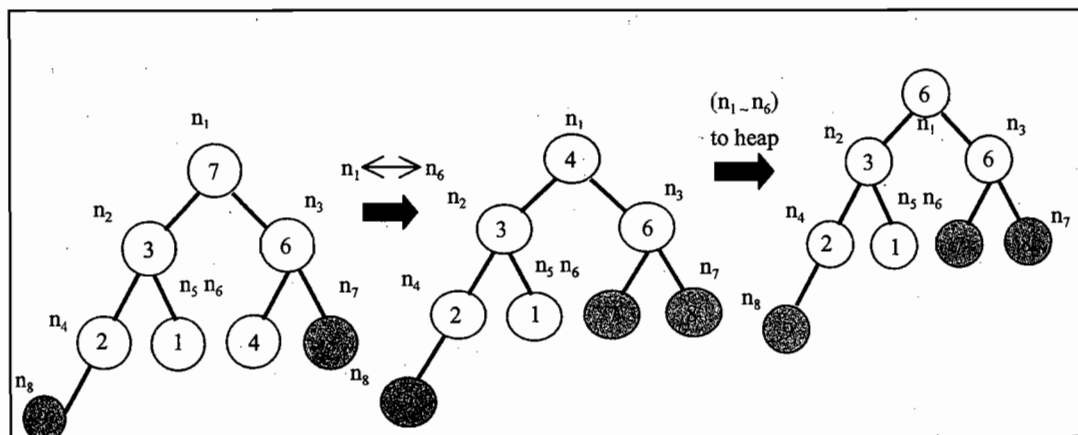
<1>



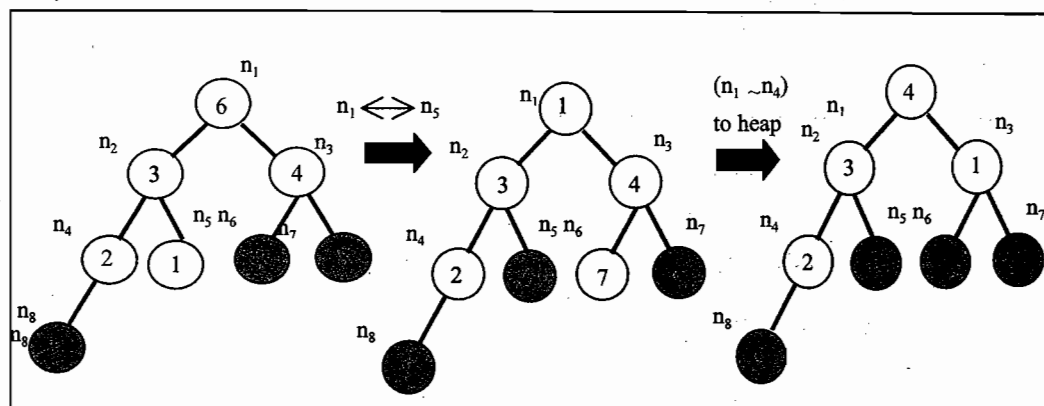
<2>



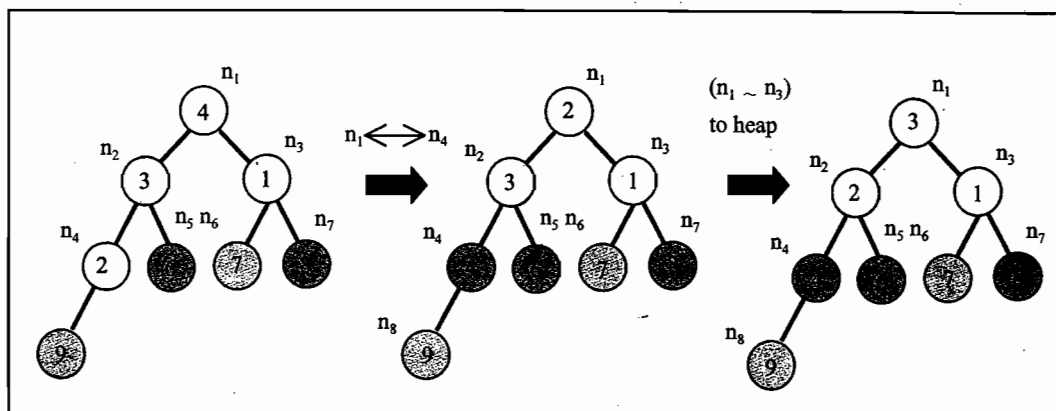
<3>



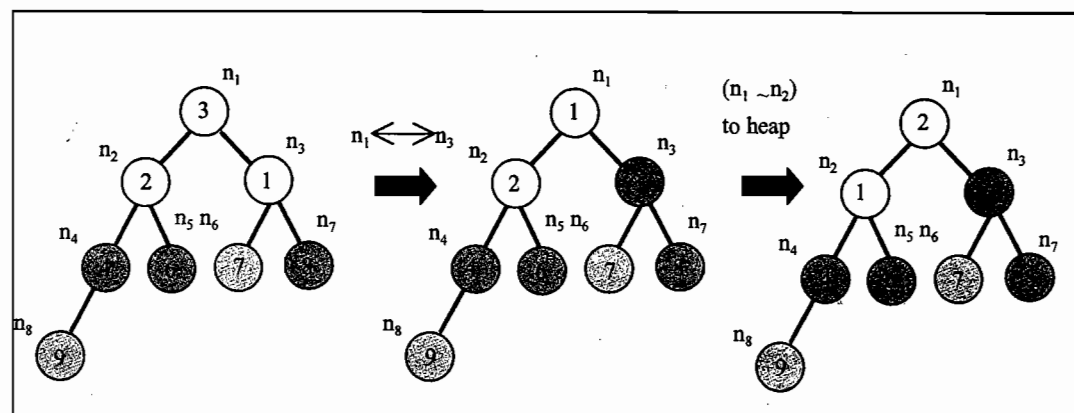
&lt;4&gt;



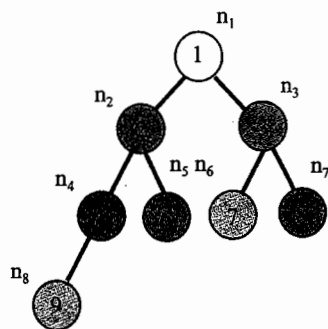
&lt;5&gt;



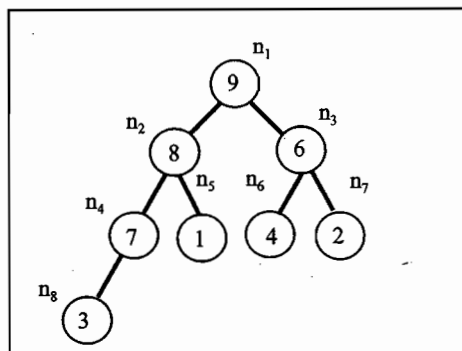
&lt;6&gt;



&lt;7&gt;



完成堆排序



相对应的数组结构排列:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

累堆排序法的特性为不稳定性排序。因为每次节点在做交换时, 需要一个额外的暂存空间, 故空间复杂度为  $O(1)$ 。而累堆为树状结构, 时间复杂度和树的高度有关, 故为  $O(n \log_2 n)$ 。

#### 程序实例:

使用累堆排序法设计一个排序程序。

#### 程序构思:

1. 读取数值存入二叉树数组 list 中
2. 将二叉树转成最大堆
3. 堆的最大值和数组最后一个数值交换
4. 其余数值进行堆重建, 并打印目前排序结果
5. 重复 3、4, 直到所有值均已排序完成
6. 打印最终排序结果

#### 程序源代码:

```

01 /* ===== Program Description===== */
02 /* 程序名称: Sort_04.c */
03 /* 程序目的: 使用累堆排序法设计一个排序程序 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /* ----- */

```

```

07 /* 建立累堆 */
08 /*-----*/
09 void createHeap(int *heap,int root, int index)
10 { int i, j ,
11 int temp; /*子数值交换时的暂存变量*/
12 int finish; /*判断累堆是否建立完成*/
13
14 j=2*root; /*子节点的 index*/
15 temp=heap[root]; /*暂存 heap 的 root 值*/
16 finish=0; /*默认累堆建立尚未完成*/
17
18 while (j<= index && finish==0)
19 {
20 /******找最大的子节点******/
21 if (j < index)
22 if (heap[j] < heap[j+1])
23 j++;
24
25 if (temp >= heap[j])
26 finish=1; /*累堆建立完成*/
27 else
28 {
29 heap[j/2]=heap[j]; /*父节点=目前节点*/
30 j=2*j;
31 }
32 heap[j/2]=temp; /*父节点=root 值*/
33 }
34
35 /*-----*/
36 /* 进行累堆排序 */
37 /*-----*/
38 void HeapSort(int *heap, int index)
39 { int i, j, temp;
40
41 /******将二叉树转成 heap******/
42 for (i= (index/2); i >= 1; i--)
43 createHeap(heap, i , index);
44
45 /******开始进行累堆排序******/
46 for (i = index-1; i >= 1; i--)
47 {
48 temp=heap[i+1]; /*heap 的 root 值和最后一个值交换*/
49 heap[i+1]=heap[i];
50 heap[i]=temp;
51
52 createHeap(heap,1,i); /*对其余数值重建累堆*/
53
54 printf("Sorting process:"); /*打印累堆的处理过程*/
55 for (j=1; j <= index; j++)
56 printf("%d ", heap[j]);
57 printf("\n");
58 }
59 }
60
61 /*-----*/
62 /***主程序:输入数值数据=>进行累堆排序=>打印排序结果***/
63 /*-----*/
64 void main()
65 { int list[20]; /*设置数组最大长度为 20*/
66 int node; /*读入输入值所使用的暂存变量*/

```

```

68 int i ,index,
69
70 printf("\n Please input the values you want to sort(Exit for 0):\n");
71
72 list[0]=0;
73 index=1;
74
75 /*****读取数值存入数组中*****/
76 scanf("%d", &node); /*读输入值存到变量 node*/
77 while (node != 0) /*数列尚未结束*/
78 {
79 list[index]=node;
80 index=index+1;
81 scanf("%d",&node);
82 }
83 index--;
84
85 printf("Source values:");
86 for (i=1;i<=index;i++)
87 printf("%d ",list[i]);
88 printf("\n\n");
89
90 /*****进行累堆排序*****/
91 HeapSort (list, index);
92
93 /*****打印最终排序结果*****/
94 printf("\nSorting result:");
95 for (i=1; i <= index; i++)
96 printf("%d ", list[i]);
97 printf("\n");
98 }

```

运行结果:

```

C:\DS>Sort_04
Please input the values you want to sort(Exit for 0):
89 23 82 48 21 46 11 18 0

Source values: 89 23 82 48 21 46 11 18 0

Sorting process: 82 48 46 23 21 19 11 89
Sorting process: 48 23 46 11 21 19 82 89
Sorting process: 46 23 19 11 21 48 82 89
Sorting process: 23 21 19 11 46 48 82 89
Sorting process: 21 11 19 23 46 48 82 89
Sorting process: 19 11 21 23 46 48 82 89
Sorting process: 11 19 21 23 46 48 82 89

Sorting result: 11 19 21 23 46 48 82 89

C:\DS>

```

## 8.4 内部排序法——插入式排序

内部排序法中的插入式排序, 是对于欲排序的元素以插入的方式找寻该元素的适当位置, 以达到排序的目的。

插入式排序法又可分为两种:

1. 插入排序法 (Insertion Sort)
2. 谢耳排序法 (Shell Sort)
3. 二叉树排序法 (Binary-tree Sort)

### 8.4.1 插入排序法

方法:

假设欲排序的数组元素  $V_1, V_2, V_3, V_4$  且  $V_3 > V_2 > V_4 > V_1$ 。依序进行插入元素的动作, 在每次插入时该元素会放在适当的排序位置, 直到最后一个元素插入后, 则所有元素排序完成。

起始数组:

|       |       |       |       |
|-------|-------|-------|-------|
| $V_1$ | $V_2$ | $V_3$ | $V_4$ |
|-------|-------|-------|-------|

插入 " $V_2$ ":

|  |  |       |       |
|--|--|-------|-------|
|  |  | $V_3$ | $V_4$ |
|--|--|-------|-------|

插入 " $V_3$ ":

|  |  |  |       |
|--|--|--|-------|
|  |  |  | $V_4$ |
|--|--|--|-------|

插入 " $V_4$ ":

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|--|--|--|--|

举例说明:

欲排列数值 6, 9, 3, 4, 1, 5

起始数组:

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 6 | 9 | 3 | 4 | 1 | 5 |
|---|---|---|---|---|---|

插入 "9":  $9 > 6$ , 故 9 置于 6 之后

|  |  |   |   |   |   |
|--|--|---|---|---|---|
|  |  | 3 | 4 | 1 | 5 |
|--|--|---|---|---|---|

插入 "3":  $3 < 6$ , 故 6, 9 往后移, 3 置于原 6 的位置

|  |  |  |   |   |   |
|--|--|--|---|---|---|
|  |  |  | 4 | 1 | 5 |
|--|--|--|---|---|---|

插入 "4":  $3 > 4 > 6$ , 故 6, 9 往后移, 4 置于原 6 的位置

|  |  |  |  |   |   |
|--|--|--|--|---|---|
|  |  |  |  | 1 | 5 |
|--|--|--|--|---|---|



```

15 insertnode=list[i]; /*设置欲插入的数值*/
16 j = i -1; /*欲插入数组的开始位置*/
17 while (j >= 0 && insertnode < list[j]) /*找适当的插入位置*/
18
19 list [j+1] = list [j];
20 j--;
21 }
22 list[j+1]=insertnode; /*将数值插入*/
23
24 /*****打印目前排序结果*****/
25 printf("\n Current sorting result:");
26 for (k=0; k<index; k++)
27 {
28 printf("%d ",list[k]);
29 }
30 }
31 }
32
33 /*-----*/
34 /* 主程序:输入数值数据→进行插入排序→打印排序结果 */
35 /*-----*/
36 void main()
37 { int list[20]; /*设置数组最大长度为20*/
38 int node; /*读入输入值所使用的暂存变量*/
39 int i,index,
40
41 printf("\n Please input the values you want to sort(Exit for 0):\n");
42
43 index=0;
44
45 /*****读取数值存入数组中*****/
46 scanf("%d", &node); /*读输入值存到变量node*/
47 while (node != 0) /*数列尚未结束*/
48 {
49 list[index]=node;
50 index=index+1;
51 scanf("%d",&node);
52 }
53
54 /*****进行插入排序*****/
55 InsertSort(list,index);
56
57 /*****打印最终排序结果*****/
58 printf("\n Final sorting result:");
59 for (i=0;i<index;i++)
60 { printf("%d ",list[i]);
61 }

```

运行结果:

```

C:\DS>Sort_05
Please input the values you want to sort(Exit for 0):
76 32 12 45 22 57 19 61 0

Current sorting result:32 76 12 45 22 57 19 61
Current sorting result:12 32 76 45 22 57 19 61
Current sorting result:12 32 45 76 22 57 19 61
Current sorting result:12 22 32 45 76 57 19 61
Current sorting result:12 22 32 55 57 76 19 61
Current sorting result:12 19 22 32 45 57 76 61
Current sorting result:12 19 22 32 45 57 61 76

```



```
Final sorting result:12 19 22 32 45 57 61 76
C:\DS>
```

## 8.4.2 谢耳排序法

方法:

将欲排序的数值依某个间隔长度分成数个数列集合,再针对各个数列集合进行“插入法排序”,重复进行数列分割,每次分割的间隔长度缩小为上一次分割间隔长度的二分之一,直到分割间隔为零停止,则数列排序完成。

举例说明:

|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| data | 12 | 57 | 48 | 33 | 25 | 92 | 86 | 37 |

步骤 1:  $\text{length} = 8 / 2 = 4$

|      |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  |    |
| data | 25 | 57 | 48 | 37 | 12 | 92 | 86 |
| (1)  |    |    |    |    |    |    |    |
| (2)  |    |    |    |    |    |    |    |
| (3)  |    |    |    |    |    |    |    |
| (4)  |    |    |    |    |    |    |    |

针对集合(1)(2)(3)(4)分别进行“插入法排序”

|      |           |    |    |           |           |    |    |           |
|------|-----------|----|----|-----------|-----------|----|----|-----------|
|      | 0         | 1  | 2  | 3         | 4         | 5  | 6  | 7         |
| data | <b>12</b> | 57 | 48 | <b>33</b> | <b>25</b> | 92 | 86 | <b>37</b> |

步骤 2:  $\text{length} = 4 / 2 = 2$

|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| data | 12 | 57 | 48 | 33 | 25 | 92 | 86 | 37 |
| (1)  |    |    |    |    |    |    |    |    |
| (2)  |    |    |    |    |    |    |    |    |

针对集合(1) (2)分别进行“插入法排序”

步骤 3:  $\text{length} = 2 / 2 = 1$

|      |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| data | 12 | 33 | 25 | 37 | 48 | 57 | 86 | 92 |

(1) 

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

针对集合(1)进行“插入法排序”

|      |    |           |           |    |    |    |    |    |
|------|----|-----------|-----------|----|----|----|----|----|
|      | 0  | 1         | 2         | 3  | 4  | 5  | 6  | 7  |
| data | 12 | <b>25</b> | <b>33</b> | 37 | 48 | 57 | 86 | 92 |

谢耳排序法的优点是以插入的方式进行排序，方法简易。由于插入排序法对已排序好的部分会快速处理，故最后几次程序速度会较快。谢耳排序法的空间复杂度为  $O(1)$ ，而时间复杂度为  $O(n^2)$ ，其中  $1 < r < 2$ 。

程序实例：

使用谢耳排序法设计一个排序程序。

程序构思：

1. 读取数值存入数组 list 中
2. 默认集合的数值位置间隔  $\text{length} = \text{数值个数} / 2$
3. 将数值列依  $\text{length}$  分割成数个集合
4. 分别对各个集合进行“插入法排序”
5. 预备下次数值列分割
  - $\text{length} = \text{length} / 2$  (缩小集合范围为原范围之  $1/2$ )
  - if  $\text{length} \neq 0$  then
  - 回到(2)继续做数值列分割、排序
  - else
  - 排序完成

程序源代码：

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Sort_06.c */
03 /* 程序目的: 使用谢耳排序法设计一个排序程序 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 谢耳排序法的处理 */
08 /*-----*/
09 void ShellSort(int *list, int index)
10 {
11 int i, j, k,
12 int change; /*记录数值是否有交换位置*/
13 int temp; /*于数值交换时的暂存变量*/

```

```

13 int length; /*分割集合的间隔长度*/
14 int process; /*进行处理的位置*/
15
16 length=index / 2; /*初始集合间隔长度*/
17
18 while (length != 0) /*数值列仍可进行分割*/
19 {
20 /*对各个集合进行处理*/
21 for (j = length; j < index ; j++)
22 {
23 change=0;
24 temp=list[j]; /*暂存 list[j] 的值,待交换值时用*/
25 process=j - length; /*计算进行处理的位置*/
26
27 /*进行集合内数值的比较与交换值*/
28 while (temp < list[process] && process >= 0 && j <= index)
29 {
30 list[process+length]=list[process];
31 /*计算下一个欲进行处理的位置*/
32 process=process-length;
33 change=1;
34 }
35
36 list[process+length]=temp; /*与最后的数值交换*/
37
38
39 if (change !=0)
40 {
41 /******打印目前排序结果******/
42 printf("\n Current sorting result:");
43 for (k=0;k<index;k++)
44 {
45 printf("%d ",list[k]);
46 }
47 }
48 length=length / 2; /*计算下次分割的间隔长度*/
49 }
50 }
51
52
53 /*-----*/
54 /** *主程序:输入数值数据=>进行谢耳排序=>打印排序结果** */
55 /*-----*/
56 void main()
57 { int list[20]; /*设置数组最大长度为 20*/
58 int node; /*读入输入值所使用的暂存变量*/
59 int i,index;
60
61 printf("\n Please input the values you want to sort(Exit for 0):\n");
62
63 index=0;
64
65 /*-----读取数值存到数组中-----*/
66 scanf("%d", &node); /*读输入值存到变量 node*/
67 while (node != 0) /*数列尚未结束*/
68 {
69 list[index]=node;
70 index=index+1;
71 scanf("%d",&node);
72 }
73

```

```

74 /*****进行谢耳排序*****/
75 ShellSort(list,index);
76
77 /*-----打印最终的排序结果-----*/
78 printf("\n Final sorting result:");
79 for (i=0;i<index;i++)
80 {
81 printf("%d ",list[i]);
82 }
83 }

```

运行结果:

```

C:\DS>Sort_06
Please input the values you want to sort(Exit for 0):
25 57 48 37 12 92 86 33 0

Current sorting result: 12 57 48 37 25 92 86 33
Current sorting result: 12 57 48 33 25 92 86 37
Current sorting result: 12 33 48 57 25 92 86 37
Current sorting result: 12 33 25 57 48 92 86 37
Current sorting result: 12 33 25 37 48 57 86 92
Current sorting result: 12 25 33 37 48 57 86 92

Final sorting result: 12 25 33 37 48 57 86 92

C:\DS>

```

### 8.4.3 二叉树排序法

在插入式排序中,前两节所介绍的“插入排序法”和“谢耳排序法”是使用数组结构进行排序。本节介绍的“二叉树排序法”将使用树状结构来进行排序处理。

方法:

1. 将欲排序的元素一一以建立二叉树的方式插入,建立二叉树需满足二个条件:
  - (1) 每一个节点最多只有两个子节点 (左节点、右节点)
  - (2) 若一节点有子节点,则该节点的数据要比左节点的数据大,且比右节点的数据小 (左节点 < parent < 右节点)
2. 使用二叉树中序遍历的方式将二叉树的节点打输出来,即可得到元素的排序结果。

举例说明:

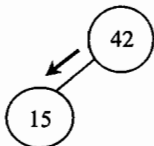
欲排列数组中的元素

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 42 | 15 | 61 | 22 | 54 | 38 |
|----|----|----|----|----|----|

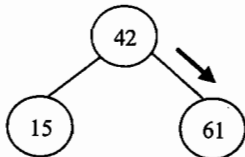
(1) 读入“42”，为二叉树的树根



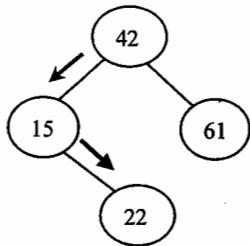
(2) 读入“15”， $15 < 42$ ，故插入为“42”的左节点



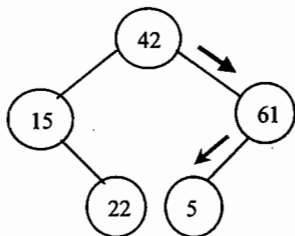
(3) 读入“61”， $61 > 42$ ，故插入为“42”的右节点



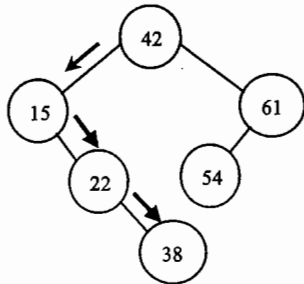
(4) 读入“22”， $22 < 42$ ，往“42”的左子树走  
 $22 > 15$ ，故插入为“15”的右节点



(5) 读入“54”， $54 > 42$ ，往“42”的右子树走  
 $54 < 61$ ，故插入为“61”的右节点



(6) 读入“38”， $38 < 42$ ，往“42”的左子树走  $38 > 15$ ，往“15”的右子树走  $38 > 22$ ，故插入为“22”的右节点



(7) 用二叉树中序遍历的方式，可得排序结果

15, 22, 38, 42, 54, 61

二叉树排序法的优点是以插入的方式进行排序，方法简易。缺点是，在插入节点时，第一个插入的元素必为树根，若该值在欲排序的元素中偏大或偏小，则易造成二叉树的歪斜程度愈大。若插入二叉树条件为(左节点  $<$  parent  $\leq$  右节点)，则为稳定性排序。若插入的条件为(左节点  $\leq$  parent  $<$  右节点)，则是不稳定性排序。空间复杂度为  $O(n)$ ，而时间复杂度受树的高度影响为  $O(n * \log_2 n)$ 。

程序实例：

使用二叉树排序法设计一个排序程序。

## 程序构思:

1. 读取数值存入数组 list 中
2. 清除二叉树数组 btree 为 0
3. 根据 list 中的数值建立二叉树存入数组 btree 中
4. 使用“二叉树中序遍历”将最终排序结果打输出来

## 程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Sort_07.c */
03 /* 程序目的: 使用二叉树排序法设计一个排序程序 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 建立二叉树 */
08 /*-----*/
09 void binarytree(int *btree,int *list,int index)
10 {
11 int level; /*树的阶层*/
12
13 btree[1]=list[1]; /*产生二叉树的 root*/
14
15 for (i=2; i <= index; i++)
16 {
17 level=1; /*第一阶层*/
18 while (btree[level] != 0) /*判断有没有 suB.tree*/
19 {
20 /*判断要插在右子树或是左子树*/
21 if (list[i] > btree[level])
22 level=2*level+1; /*大者插在右子树*/
23 else
24 level=2*level; /*小者插在左子树*/
25 }
26 btree[level]=list[i]; /*存入节点数值*/
27 }
28 }
29
30 /*-----*/
31 /* 二叉树中序遍历(用以打印二叉树) */
32 /*-----*/
33 void inorder(int *btree, int position)
34 {
35 if (btree[position] != 0 && position < 20)
36 {
37 inorder(btree, 2*position); /*左子树*/
38 if (btree[position] != 0)
39 printf("%d ", btree[position]); /*打印节点的值*/
40 inorder(btree, 2*position+1); /*右子树*/
41 }
42 }
43
44 /*-----*/
45 /***主程序:输入数值数据=>进行二叉树排序=>打印排序结果** */
46 /*-----*/
47 void main()
48 {
49 int list[20]; /*设置数组最大长度为 20*/
50 int btree[2048]; /*默认存 binary tree 的数组长度*/
51 int node; /*读入输入值所使用的暂存变量*/
52 int i ,index;

```

```

52
53 printf("\n Please input the values you want to sort(Exit for 0):\n");
54
55 index=1;
56
57 /*****读取数值存入数组中*****/
58 scanf("%d", &node); /*读输入值存到变量 node*/
59 while (node != 0) /*数列尚未结束*/
60 {
61 list[index]=node;
62 index=index+1;
63 scanf("%d",&node);
64 }
65 index--;
66
67 /*****清除 binary tree 数组 btree[]为 0*****/
68 for (i=0; i <= 2048; i++)
69 btree[i]=0;
70
71
72 /*****建立二叉树*****/
73 binarytree(btree, list, index);
74
75 /*****打印最终排序结果*****/
76
77 printf("\n Sorting result:");
78 inorder(btree,1);
79 printf("\n");
80 }

```

运行结果:

```

C:\DS>Sort_07
Please input the values you want to sort(Exit for 0):
45 25 69 15 24 36 98 44 28 6 0

Sorting result: 6 15 24 25 28 36 44 45 69 98

C:\DS>

```

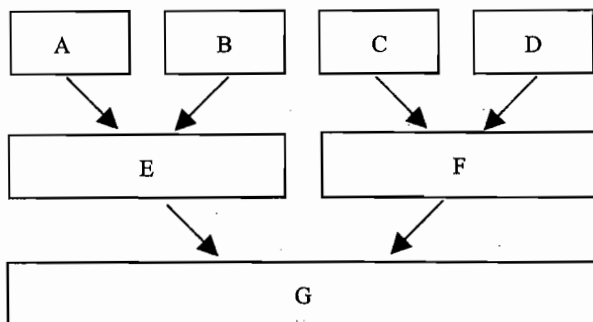
## 8.5 外部排序——合并排序法

合并排序法是外部排序最常使用的排序方法。若数据量太大无法一次完全加载内存,可使用外部辅助内存来处理排序数据,主要应用在文件排序。

方法:

将欲排序的数据分别存在数个文件大小可加载内存的文件中,再针对各个文件分别使用“内部排序法”将文件中的数据排序好写回文件。再对所有已排序好的文件两两合并,直到所有文件合并成一个文件后,则数据排序完成。

假设有4个文件A、B、C、D，其内部数据均已排序完成，则文件合并排序方式如下：



- (1) 将已排序好的A、B合并成E，C、D合并成F。  
→E、F的内部数据分别均已排好序
- (2) 将已排序好的E、F合并成G  
→G的内部数据已排好序
- (3) 4个文件A、B、C、D数据排序完成

举例说明：

欲排序的数据如下：

|      |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| data | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

步骤1: length=1

|      |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| data | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|      |   |   |   |   |   |   |   |   |

两两合并 [8, 7]、[6, 5]、[4, 3]、[2, 1]

|      |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| data | 7 | 8 | 5 | 6 | 3 | 4 | 1 | 2 |

步骤2: length=2

|      |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| data | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|      |   |   |   |   |   |   |   |   |



两两合并 [(7,8), (5,6)], [(3,4), (1,2)]

|      |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| data | 5 | 6 | 7 | 8 | 1 | 2 | 3 | 4 |

步骤 3: length=4

|      |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| data | 5 | 6 | 7 | 8 | 1 | 2 | 3 | 4 |

两两合并 [(5,6,7,8), (1,2,3,4)]

|      |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| data | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

合并排序法的优点是可用来处理大量数据的排序,是属于稳定性排序,空间复杂度(Space complexity)是  $O(n)$ , 时间复杂度(Time complexity) $O(n \log_2 n)$ 。

程序实例:

使用合并排序法设计一个排序程序。

程序构思:

1. 读取数值存入数组 list1 中
2. 清除输出数组 list2 为 0
3. 进行合并排序
  - 两两合并, 直到全部合并完成
4. 将最终排序结果打印出来

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: Sort_08.c */
03 /* 程序目的: 使用合并排序法设计一个排序程序 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 合并 2 个已排序好的数值数组 */
08 /*-----*/
09 void MergeTwo (int *source, int *distination, int left, int middle, int n)
10 { int i,j,k,t;
11
12 i=left; k=left; j=middle+1; /*设置数组指针*/
13
14 /*两个欲合并的数组均还有值尚未处理*/
15 while ((i<=middle) && (j<=n))
16 {

```

```

17 if (source[i]<=source[j]) /*将较小者先存到输出数组 destination*/
18 {
19 distination[k]=source[i];
20 i++;
21 }
22 else
23 {
24 distination[k]=source[j];
25 j++;
26 }
27 k= ++;
28 }
29
30 /*将尚未处理完的数组数值依序存入输出数组 distination 中*/
31 if (i > middle)
32 {
33 for (t=j ; t<=n; t++)
34 distination[k+t-j]=source[t];
35 }
36 else
37 {
38 for (t=i ;t<=middle; t++)
39 distination[k+t-i]=source[t];
40 }
41 }
42
43
44 /*-----*/
45 /* 将所有 partition array 分别两两合并 */
46 /*-----*/
47 void MergeAll (int *source, int *distination, int n,int length)
48 {
49 int i,t;
50 i=0;
51 while (i<= (n-2*length+1)) /*还有两段长度为 length 的 list 可合并*/
52 {
53 MergeTwo(source,distination,i,i+length-1,i+2*length-1);
54 i= i + 2*length;
55 }
56 if (i+length -1 < n)
57 {
58 /*合并两段 list,一段长度为 length,另一段长度不足 length*/
59 MergeTwo(source,distination,i,i+length-1,n);
60 }
61 else
62 {
63 /*将剩下一段不足 length 长的 list 中的值依序存到输出数组 distination*/
64 for (t=i; t<=n; t++)
65 distination[t]=source[t];
66 }
67
68 /******将 distination 中的值复制到 source******/
69 for (t=0; t<=n; t++)
70 source[t]=distination[t];
71
72 /******打印目前排序结果******/
73 printf(" Current sorting result: ");
74 for (i=0; i<=n; i++)
75 printf("%d ",distination[i]);
76 printf("\n");
77 }

```

```

78
79
80 /*-----*/
81 /* 主程序:输入数值数据=>进行合并排序=>打印排序结果 */
82 /*-----*/
83 void main()
84 { int list1[20],list2[20]; /*设置数组最大长度为 20*/
85 int node; /*读入输入值所使用的暂存变量*/
86 int length; /*记录合并数组的长度*/
87 int count; /*记录整个数值数组的总个数*/
88 int i ,index ;
89
90 printf("\n Please input the values you want to sort(Exit for 0):\n");
91
92 index=0;
93
94 /******读取数值存入数组中******/
95 scanf("%d", &node); /*读输入值存到变量 node*/
96 while (node != 0) /*数列尚未结束*/
97 {
98 list1[index]=node;
99 index=index+1;
100 scanf("%d",&node);
101 }
102
103 /******清除数组 list2[]为 0******/
104 for (i=0; i <= 19; i++)
105 list2[i]=0;
106
107 printf("\nSource values:");
108 for (i=0; i<index; i++)
109 printf("%d ",list1[i]);
110 printf("\n\n");
111
112
113 /******进行合并排序******/
114 length=1; /*设置初始合并的长度*/
115 count=index-1;
116
117 while (length < index) /*合并尚未结束*/
118 {
119 printf("**Merge length=%d \n",length);
120 MergeAll (list1, list2, count, length); /*将所有 list 两两合并*/
121 length=2*length; /*将 length 变成两倍*/
122 if (length < index)
123 {
124 printf("**Merge length=%d \n",length);
125 MergeAll (list2, list1, count, length);
126 length=2*length;
127 }
128 }
129
130
131 /******打印最终排序结果******/
132 printf("\n Final sorting result:");
133 for (i=0; i<index; i++)
134 printf("%d ",list1[i]);
135 printf("\n");
136 }

```

运行结果:

```
C:\DS>Sort_08
Please input the values you want to sort(Exit for 0):
9 8 7 6 5 4 3 2 1 0

Source values : 9 8 7 6 5 4 3 2 1
*Merge length=1
Current sorting result: 8 9 6 7 4 5 2 3 1
*Merge length=2
Current sorting result: 6 7 8 9 2 3 4 5 1
*Merge length=4
Current sorting result: 2 3 4 5 6 7 8 9 1
*Merge length=8
Current sorting result: 1 2 3 4 5 6 7 8 9

Final sorting result: 1 2 3 4 5 6 7 8 9

C:\DS>
```

## 8.6 排序法的效率比较

| 排序法    | 最差时间复杂度               | 平均时间复杂度               | 稳定度  | 空间复杂度                   |
|--------|-----------------------|-----------------------|------|-------------------------|
| 冒泡排序法  | $O(n^2)$              | $O(n^2)$              | 稳定性  | $O(1)$                  |
| 快速排序法  | $O(n^2)$              | $O(n \cdot \log_2 n)$ | 不稳定性 | $O(\log_2 n) \sim O(n)$ |
| 选择排序法  | $O(n^2)$              | $O(n^2)$              | 稳定性  | $O(1)$                  |
| 累堆排序法  | $O(n \cdot \log_2 n)$ | $O(n \cdot \log_2 n)$ | 不稳定性 | $O(1)$                  |
| 插入排序法  | $O(n^2)$              | $O(n^2)$              | 稳定性  | $O(1)$                  |
| 谢耳排序法  | $O$                   | $O$                   | 不稳定性 | $O(1)$                  |
| 二叉树排序法 | $O(n^2)$              | $O(n \cdot \log_2 n)$ | 不一定  | $O(n)$                  |
| 外部排序法  | $O(n \cdot \log_2 n)$ | $O(n \cdot \log_2 n)$ | 稳定性  | $O(n)$                  |

### 【习题】

一、复习:

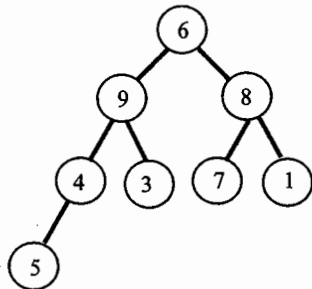
- 何谓“排序”?
- 试说明“稳定性排序”和“不稳定性排序”。
- 利用冒泡排序法处理  $n+1$  个数据, 最快比较 \_\_\_\_\_ 次后可完成排序, 最慢比较 \_\_\_\_\_ 次后可完成排序。
- 关于外部排序和内部排序的差异, 下列哪一个正确?
  - 内部排序的处理速度较慢
  - 外部存储装置容量较大
  - 数据处理机制不同

(d) 外部存储装置可以随机存取数据

5. 说明何谓“外部排序”、“内部排序”？并各举两个排序法。

二、应用：

1. 试说明何谓“谢耳排序法”？
2. 试说明何谓“合并排序法”？
3. 试绘出下列二叉树转换成最大累堆树的过程：



4. 请写出使用冒泡排序法对下列数值进行排序的过程：  
79, 84, 65, 43, 86, 92, 80, 98, 75
5. 请写出使用合并排序法对下列数值进行排序的过程：  
23, 44, 12, 15, 56, 17, 66, 31  
(使用[] 标示出每次合并的集合)
6. 试分别写出下列排序法，排序结果为“由大到小”的程序。
  - (1) 冒泡排序法
  - (2) 选择排序法
  - (3) 累堆排序法
  - (4) 谢耳排序法

# 查 找

## 第 9 章

- ◆ 何谓查找
- ◆ 线性查找
- ◆ 折半查找
- ◆ 费氏查找
- ◆ 插补查找
- ◆ 杂凑查找
- ◆ 二叉查找树

## 9.1 何谓查找

查找(Search)的目的在于从一些数据中寻找出一个特定的值。例如:我们从电话簿中寻找出一个朋友的电话,从全班成绩表中寻找出某一位同学的成绩。从电话簿中寻找出一个朋友的电话时,我们必须先确定要想要找的朋友姓名,此时朋友的姓名就是所谓的“键值”(Key),我们再以朋友的姓名来和电话簿中的姓名数据比较,当我们找到朋友的姓名时,便可得到朋友的电话。

当然如果电话簿上的数据并没有依姓氏的笔划来排序,我们可能要从第一笔开始找,直到找到该姓名才能得到其电话,如果电话簿中并没有此位朋友的电话,那我们可能要把整个电话簿中的数据全都找完才知道,这种作法是不是很没有效率。如果我们把电话簿的数据依姓氏及名字排序,我们寻找朋友电话时,也只要从朋友姓氏所对映的页次开始找就可以了,如果一来不仅省时,也增加每次寻找数据的效率。

数据排序的方法,我们在排序那一章就有提到。这一章我们要讨论一些常用的数据查找方法。

1. 线性查找
2. 折半查找
3. 费氏查找
4. 插补查找
5. 杂凑查找
6. 二叉查找树

其中线性查找可以用在经过排序的数据查找,而其它的查找方法则需要先将数据先排序完成才有一定的顺序可循,再按一定的规则进行查找工作。

## 9.2 线性查找

线性查找法(Linear Searching)又称为循序式查找(Sequential Searching),是从数据中的第一笔数据开始查找比较,如果找到则返回该值或该位置,如果没有找到则往下一笔数据查找比较,直到查找到最后一笔数据为止。

例如,有一个整数数组的数据内容如下:

|      | 0  | 1  | 2  | 3  | 4  | 5  |
|------|----|----|----|----|----|----|
| Data | 13 | 25 | 16 | 23 | 57 | 66 |

如果我们现在想找出 57, 则:

- 步骤 1: 欲寻找值 57, 与数组中第一笔数据比较。57 $\neq$ 13, 寻找下一笔。  
步骤 2: 欲寻找值 57, 与数组中第二笔数据比较。57 $\neq$ 25, 寻找下一笔。  
步骤 3: 欲寻找值 57, 与数组中第 3 笔数据比较。57 $\neq$ 16, 寻找下一笔。  
步骤 4: 欲寻找值 57, 与数组中第 4 笔数据比较。57 $\neq$ 23, 寻找下一笔。  
步骤 5: 欲寻找值 57, 与数组中第 5 笔数据比较。57 = 57, 找到该数据。

如果我们现在想找出 27, 则:

- 步骤 1: 欲寻找值 27, 与数组中第一笔数据比较。27 $\neq$ 13, 寻找下一笔。  
 步骤 2: 欲寻找值 27, 与数组中第二笔数据比较。27 $\neq$ 25, 寻找下一笔。  
 步骤 3: 欲寻找值 27, 与数组中第 3 笔数据比较。27 $\neq$ 16, 寻找下一笔。  
 步骤 4: 欲寻找值 27, 与数组中第 4 笔数据比较。27 $\neq$ 23, 寻找下一笔。  
 步骤 5: 欲寻找值 27, 与数组中第 5 笔数据比较。27 $\neq$ 57, 已到最后一笔, 仍未找到数据, 所以返回未找到数据。

#### 程序实例:

设计一个线性查找的程序。

#### 程序构思:

运用线性查找的概念。

如果找到则输出该数据的位置及内容。

如果没有找到则寻找下一笔数据, 直到最后一笔为止。

#### 程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: 1_search.c */
03 /* 程序目的: 设计一个线性查找的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 int Data[20] = { 12, 76, 29, 22, 15,
07 62, 29, 58, 35, 67,
08 58, 33, 28, 89, 90,
09 28, 64, 48, 20, 77}; /* 数据数组 */
10 int Counter = 1;
11 /* ----- */
12 /* 顺序查找 */
13 /* ----- */
14 int Linear_Search(int Key)
15 {
16 int Index = 0; /* 计数变量 */
17
18 while (Index < 20)
19 {
20 if (Key == Data[Index]) /* 查找到数据时 */
21 {
22 printf("Data[%d] = %d \n", Index, Key);
23 return 1;
24 }
25 Index++;
26 Counter++;
27 }
28 return 0;
29 }
30
31 /* ----- */
32 /* 主程序 */
33 /* ----- */
34 void main ()
35 {
36 int KeyValue; /* 欲查找数据变量 */
37

```



```

38 printf("Please enter your key value : ");
39 scanf("%d",&KeyValue);
40
41 if (Linear_Search(KeyValue))
42 printf("Search Time = %d\n",Counter); /* 输出查找次数 */
43 else
44 printf("No Found!!\n"); /* 输出没有找到数据 */
45 }

```

运行结果:

```

C:\DS>l_search
Please enter your key value : 99
No Found!!

C:\DS>l_search
Please enter your key value : 58
Data[7] = 58
Search Time = 8

C:\DS>

```

以上的范例是采用将欲查找数据与数组中的数据进行比较,但是每次除了比较数据还得要判断是否已经达到数组的边界(数组最后一笔),如上述程序的第 18 行,这样反而让程序变得没有效率。如果换一个方式,我们将欲查找数据放在最后一笔数据之后,如果查找到数据,再来判断是否是我们之前所存储的那个位置,如果是的话,表示未能查找到数据,反之,则表示查找到数据。注意这种方式只增快查找速度而已,但是时间复杂度上并没有比范例好。

线性查找法的最佳状态时间复杂度为  $B(n) = 1 \in O(n)$ 。表示第一次就找到数据。而最坏状态的时间复杂度为  $W(n) = n \in O(n)$ 。表示未能找到数据或数据出现在最后一笔。

其平均状态的时间复杂度为  $A(n) = (1 + 2 + \dots + n) / n = (n + 1) / 2 \in O(n)$

程序实例:

采用将欲查找数据放在最后一笔数据之后重新设计线性查找法。

程序构思:

运用线性查找的概念。

如果找到则判断是否是之前欲查找数据所存储的位置,如果是的话,则表示未能查找到数据。否则的话,表示查找到数据,则输出该数据的位置及内容。

如果没有找到则寻找下一笔数据。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: s_search.c */
03 /* 程序目的: 采用将欲查找数据放在最后一笔数据之后重新 */
04 /* 设计线性查找法。 */
05 /* Written By Kuo-Yu Huang. (WANT Studio.) */
06 /* ===== */
07 #define Max 20
08 int Data[Max+1] = { 12, 76, 29, 22, 15,
09 62, 29, 58, 35, 67,
10 58, 33, 28, 89, 90,

```

```

11 28, 64, 48, 20, 77}; /* 数据数组 */
12 int Counter = 1;
13 /* ----- */
14 /* 顺序查找 */
15 /* ----- */
16 int Linear_Search(int Key)
17 {
18 int Index = 0; /* 计数变量 */
19
20 while (1)
21 {
22 if (Key == Data[Index]) /* 查找到数据时 */
23 {
24 if (Index != Max)
25 {
26 printf("Data[%d] = %d \n",Index,Key);
27 return 1;
28 }
29 else
30 return 0;
31 }
32 Index++;
33 Counter++;
34 }
35 }
36
37 /* ----- */
38 /* 主程序 */
39 /* ----- */
40 void main ()
41 {
42 int KeyValue; /* 欲查找数据变量 */
43
44 printf("Please enter your key value : ");
45 scanf("%d",&KeyValue);
46
47 Data[Max] = KeyValue; /* 将欲查找数据存放在最后一笔数据之后 */
48
49 if (Linear_Search(KeyValue))
50 printf("Search Time = %d\n",Counter); /* 输出查找次数 */
51 else
52 printf("No Found!!\n"); /* 输出没有找到数据 */
53 }

```

运行结果:

```

C:\DS>s_search
Please enter your key value : 99
No Found!!

C:\DS>s_search
Please enter your key value : 58
Data[7] = 58
Search Time = 8

C:\DS>

```

## 9.3 折半查找

当然读者一定发现,对于已经排序好的数据,如果用线性查找法来查找一定很浪费时间,因为如果我们已经知道,我们要查找的数据是值最大的元素,我们只要找最后一个就好了。同样的道理,如果我们在英文字典中找“structure”这个单字,我们一翻字典翻到前缀为“m”的部分,我们的直觉一定往后翻,而不是往前翻,因为“structure”这个单字的前缀为“s”,而“s”在英文字母的排列中排在“m”之后,所以我们不必从前缀“a”的部分一页一页的翻。这就是已经排序过的数据对查找的好处。

同理,折半查找法(Binary Searching)就是运用这种方法。欲查找值先与该数据(KeyValue)的中位数(middle)比较,假设数据的内容为 Data[0]到 Data[n],则  $middle = n / 2$ , 左边界 left = 0, 右边界 right = n。其原理可归纳为下列 3 点:

1. 若 KeyValue 小于 Data[middle]:  
表示 KeyValue 可能出现在 Data[middle]之前,所以查找 Data[0]到 Data[middle-1]之间的数据。  
这时  $left = left$ ,  $right = middle - 1$ , 而  $middle = (left + right) / 2$
2. 若 KeyValue 大于 Data[middle]:  
表示 KeyValue 可能出现在 Data[middle]之后,所以查找 Data[middle+1]到 Data[n]之间的数据。  
这时  $left = middle + 1$ ,  $right = right$ , 而  $middle = (left + right) / 2$
3. 若 KeyValue 等于 Data[middle]:  
表示已查找到数据。

重复执行上述 3 个步骤直到  $left = right$  或者找到欲查找数据为止。

例如:有一个整数数组的数据内容如下:

|      | 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11  |
|------|---|---|----|----|----|----|----|----|----|----|----|-----|
| Data | 5 | 7 | 12 | 25 | 34 | 37 | 43 | 46 | 58 | 80 | 92 | 105 |

如果我们现在想找出 25, 则, 此时左边界(left)= 0、右边界(right)= 11、中位数(middle)=  $11 / 2 = 5$ (取下高斯)、欲查找值(KeyValue)= 25:

步骤 1:

$$25 < \text{Data}[5] = 37$$

$$\text{左边界}(\text{left}) = 0, \text{右边界}(\text{right}) = 5 - 1 = 4$$

$$\text{中位数}(\text{middle}) = (0 + 4) / 2 = 2。$$

步骤 2:

$$25 > \text{Data}[2] = 12$$

$$\text{左边界}(\text{left}) = 2 + 1 = 3, \text{右边界}(\text{right}) = 4$$

$$\text{中位数}(\text{middle}) = (3 + 4) / 2 = 3。$$

步骤 3:

$$25 = \text{Data}[3] = 25$$

表示找到数据。

如果我们现在想找出 50, 则, 此时左边界(left)= 0、右边界(right)= 11、中位数(middle)=  $11 / 2 = 5$ (取

下高斯)、欲查找值(KeyValue)=50:

步骤 1:

$50 > \text{Data}[5] = 37$

左边界(left)=5+1=6、右边界(right)=11

中位数(middle)=(6+11)/2=8。

步骤 2:

$50 > \text{Data}[8] = 58$

左边界(left)=6、右边界(right)=8-1=7

中位数(middle)=(6+7)/2=6。

步骤 3:

$50 > \text{Data}[6] = 47$

左边界(left)=6、右边界(right)=7-1=6

中位数(middle)=(6+6)/2=6。

步骤 4:

$50 > \text{Data}[6] = 47$

左边界(left)=6 等于右边界(right)=6。未能查找到数据。

折半查找法的最佳状态时间复杂度为  $B(n) = 1 \in O(n)$ 。表示第一次就找到数据。而最坏状态的时间复杂度为  $W(n) = n \in O(\log n)$ 。表示未能找到数据或数据出现在最后一笔。可由  $T(n) = T(n/2) + 1$  求得  $T(n) = (\log_2 n) + 1 \in O(\log n)$

其平均状态的时间复杂度求法如下:

取  $n = 2^k - 1$ , 则平均检查的元素为

$$S = 1/n (1 * 1 + 2 * 2 + 3 * 2^2 + \dots + k * 2^{k-1}) \text{-----}(i)$$

同乘 2 后

$$2S = 1/n (1 * 2 + 2 * 2^2 + \dots + k * 2^k) \text{-----}(ii)$$

(ii) - (i) 得

$$S = 1/n (k * 2^k + 1 - 2 - 2^2 - \dots - 2^{k-1}) = 1/n ((n+1)\log_2(n+1) - n)$$

所以  $A(n) = 1/n ((n+1)\log_2(n+1) - n) \in O(\log n)$

以下我们先举一个用非递归方式设计折半查找法的程序。然后再举一个用递归方式设计折半查找法的程序, 让读者了解折半查找法的运用。

程序实例:

运用非递归方式设计折半查找法的程序。

程序构思:

假设  $\text{middle} = n/2$ , 此时左边界  $\text{left} = 0$ , 右边界  $\text{right} = n$ 。

如果  $\text{KeyValue} < \text{Data}[\text{middle}]$ :

表示  $\text{KeyValue}$  可能出现在  $\text{Data}[\text{middle}]$  之前, 所以查找  $\text{Data}[0]$  到  $\text{Data}[\text{middle}-1]$  之间的数据。

这时  $\text{left} = \text{left}$ ,  $\text{right} = \text{middle} - 1$ , 而  $\text{middle} = (\text{left} + \text{right}) / 2$

如果  $\text{KeyValue} > \text{Data}[\text{middle}]$ :

表示 KeyValue 可能出现在 Data[middle]之后, 所以查找 Data[middle+1]到 Data[n]之间的数据。  
这时  $left = middle + 1$ ,  $right = right$ , 而  $middle = (left + right) / 2$

如果  $KeyValue = Data[middle]$ :

表示已查找到数据。

重复执行上述 3 个步骤直到  $left = right$  或者找到欲查找数据为止。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: b_search.c */
03 /* 程序目的: 运用非递归方式设计折半查找法的程序 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #define Max 20
07 int Data[Max] = { 12, 16, 19, 22, 25,
08 32, 39, 48, 55, 57,
09 58, 63, 68, 69, 70,
10 78, 84, 88, 90, 97}; /* 数据数组 */
11 int Counter = 1; /* 计数器 */
12 /* ----- */
13 /* 二分查找法 */
14 /* ----- */
15 int Binary_Search(int Key)
16 {
17 int Left; /* 左边界变量 */
18 int Right; /* 右边界变量 */
19 int Middle; /* 中位数变量 */
20
21 Left = 0;
22 Right = Max - 1;
23
24 while (Left <= Right)
25 {
26 Middle = (Left + Right) / 2;
27 if (Key < Data[Middle]) /* 欲查找值较小 */
28 Right = Middle - 1; /* 查找前半段 */
29 else if (Key > Data[Middle]) /* 欲查找值较大 */
30 Left = Middle + 1; /* 查找后半段 */
31 else if (Key == Data[Middle]) /* 查找到数据 */
32 {
33 printf ("Data[%d] = %d\n", Middle, Data[Middle]);
34 return 1;
35 }
36 Counter++;
37 }
38 return 0;
39 }
40
41 /* ----- */
42 /* 主程序 */
43 /* ----- */
44 void main ()
45 {
46 int KeyValue; /* 欲查找数据变量 */
47
48 printf("Please enter your key value : ");
49 scanf("%d", &KeyValue);

```

```

50
51 if (Binary_Search(KeyValue))
52 printf("Search Time = %d\n",Counter); /* 输出查找次数 */
53 else
54 printf("No Found!!\n"); /* 输出没有找到数据 */
55 }

```

运行结果:

```

C:\DS>b_search
Please enter your key value : 80
No Found!!

C:\DS>b_search
Please enter your key value : 58
Data[10] = 58
Search Time = 4

C:\DS>

```

若以递归的方式重新设计折半查找法则为:

程序实例:

运用递归方式设计折半查找法的程序。

程序构思:

假设  $middle = n/2$ , 此时左边界  $left = 0$ , 右边界  $right = n$ 。

递归结束条件: 当  $left$  大于  $right$  时。

递归执行部分:

如果  $KeyValue < Data[middle]$ :

递归调用并传入( $Left, Middle-1, KeyValue$ )

如果  $KeyValue > Data[middle]$ :

递归调用并传入( $Middle-1, Right, KeyValue$ )

如果  $KeyValue = Data[middle]$ :

表示已查找到数据。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: r_search.c */
03 /* 程序目的: 运用递归方式设计折半查找法的程序 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #define Max 20
07 int Data[Max] = { 12, 16, 19, 22, 25,
08 32, 39, 48, 55, 57,
09 58, 63, 68, 69, 70,
10 78, 84, 88, 90, 97}; /* 数据数组 */
11 int Counter = 0; /* 计数器 */
12 /* ----- */
13 /* 二分查找法 */
14 /* ----- */

```

```

15 int Binary_Search(int Left,int Right,int Key)
16 {
17 int Middle; /* 中位数变量 */
18
19 Counter ++;
20 if (Left > Right)
21 return 0;
22 else
23 {
24 Middle = (Left + Right) / 2;
25 if (Key < Data[Middle]) /* 欲查找值较小 */
26 return Binary_Search(Left,Middle - 1,Key);/* 搜索前半段 */
27
28 else if (Key > Data[Middle]) /* 欲查找值较大 */
29 return Binary_Search(Middle + 1,Right,Key);/* 查找后半段 */
30 else if (Key == Data[Middle]) /* 查找到数据 */
31 {
32 printf ("Data[%d] = %d\n",Middle,Data[Middle]);
33 return 1;
34 }
35 return 0;
36 }
37
38 /* -----*/
39 /* 主程序 */
40 /* -----*/
41 void main ()
42 {
43 int KeyValue; /* 欲查找数据变量 */
44
45 printf("Please enter your key value : ");
46 scanf("%d",&KeyValue);
47
48 if (Binary_Search(0,Max-1,KeyValue))
49 printf("Search Time = %d\n",Counter);/* 输出查找次数 */
50 else
51 printf("No Found!!\n"); /* 输出没有找到数据 */
52 }

```

运行结果:

```

C:\DS>r_search
Please enter your key value : 60
No Found!!

C:\DS>r_search
Please enter your key value : 69
Data[13] = 69
Search Time = 5

C:\DS>

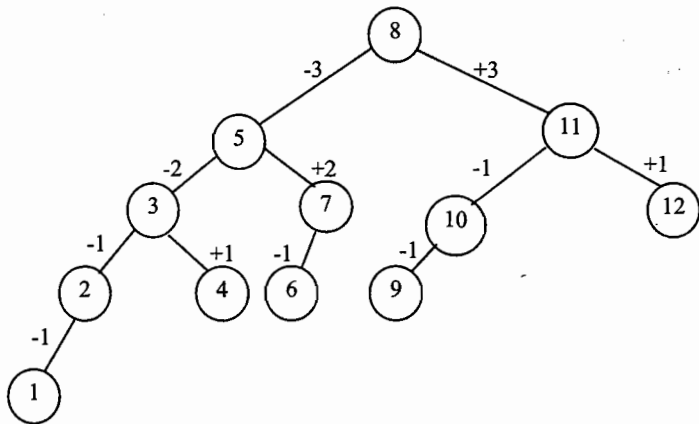
```

## 9.4 费氏查找

前一节,我们谈到了线性查找法,而这节,我们将要介绍一种与线性查找法相似的查找方法,叫做“费氏查找(Fibonacci Searching)”。线性查找法所采用的是将数据范围切半,运用到除法的运算来减少查找范围,而费氏查找法则是运算加减运算来减少范围,在计算机处理运算指令中,加减运算的效率高于乘除运算,所以费氏查找法的效率也会优于线性查找法。

在介绍费氏查找法之前,我们必须先了解什么是费氏数列?费氏数列中的元素,第0项为1、第1项为1,第3项之后的规则为“第n项为第n-1项和第n-2项的和”。按着 $F_0=1$ 、 $F_1=1$ 、 $F_n=F_{n-1}+F_{n-2}$ 的规则,所产生出的费氏数列如:1、1、2、3、5、8、13、21、34、55...

而费氏查找法就是利用费氏数列来切割数据范围的查找方法。以下我们就来介绍费氏查找法,下图是一个费氏数列的树状结构,费氏查找就是运用费氏树的特性来进行查找工作。



设有n笔已排序好的数据。每次查找时必须先找出其费氏树的树根和差值。其原则如下:

首先必须计算出费氏级数 $F(a)$ ,使得 $F(a) \leq n+1$ 。

如上图若数据有12笔,则 $F(a) \leq 12+1=13$ ,所以 $a=6$

则第一次查找费氏树的树根,即查找第 $F(A.1)$ 笔数据,第一次的差值为 $F(A.3)$ 。

从上图知 $F(A.1)=F(6.1)=8$ ,所以第一次查找第8笔数据,第一次的差值为 $F(A.3)=F(6.3)=f(3)=3$ 。

1. 若欲查找值小于第 $F(A.1)$ 笔数据的值:

表示数据在 $F(A.1)$ 笔数据之前,则我们必须查找 $F(A.1)$ 笔数据之前的数据,所以新的费氏树为原费氏树的左边子树。

2. 若欲查找值大于第 $F(A.1)$ 笔数据的值:

表示数据在 $F(A.1)$ 笔数据之后,则我们必须查找 $F(A.1)$ 笔数据之后的数据,所以新的费氏树为原费氏树的右边子树。

3. 若欲查找值等于第 $F(A.1)$ 笔数据的值:

则表示欲查找数据在 $F(A.1)$ 笔,即找到该数据。

重复执行上述3个步骤直到找到欲查找数据或差值为0为止。

例如:有一个整数数组的数据内容如下:



|      | 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11  |
|------|---|---|----|----|----|----|----|----|----|----|----|-----|
| Data | 5 | 7 | 12 | 25 | 34 | 37 | 43 | 46 | 58 | 80 | 92 | 105 |

如果我们现在想查找 7, 则此时数据 12 笔,  $F(a) = n + 1 = 12 + 1 = 13$ , 所以  $a = 6$ 。第一次查找的费氏树树根  $Root = F(A.1) = F(6.1) = F(5) = 8$ 。差值  $Distance\_1 = F(A.2) = F(6.2) = F(4) = 5$ 。差值  $Distance\_2 = F(A.3) = F(6.3) = F(3) = 3$ 。

步骤 1:

$7 < Data[8.1] = Data[7] = 46$ , 表示数据在第 8 笔数据之前。

$Root = Root - Distance\_2 = 8 - 3 = 5$

$Temp = Distance\_1 = 5$

$Distance\_1 = Distance\_2 = 3$

$Distance\_2 = Temp - Distance\_2 = 5 - 3 = 2$

步骤 2:

$7 < Data[5.1] = Data[4] = 34$ , 表示数据在第 5 笔数据之前。

$Root = Root - Distance\_2 = 5 - 2 = 3$

$Temp = Distance\_1 = 3$

$Distance\_1 = Distance\_2 = 2$

$Distance\_2 = Temp - Distance\_2 = 3 - 2 = 1$

步骤 3:

$7 < Data[3.1] = Data[2] = 12$ , 表示数据在第 3 笔数据之前。

$Root = Root - Distance\_2 = 3 - 1 = 2$

$Temp = Distance\_1 = 2$

$Distance\_1 = Distance\_2 = 1$

$Distance\_2 = Temp - Distance\_2 = 2 - 1 = 1$

步骤 4:

$7 < Data[2.1] = Data[1] = 7$

表示表示找到数据。

如果我们现在想查找 59, 此时数据 12 笔,  $F(a) \leq n + 1 = 12 + 1 = 13$ , 所以  $a = 6$ 。第一次查找的费氏树树根  $Root = F(A.1) = F(6.1) = F(5) = 8$ 。差值  $Distance\_1 = F(A.2) = F(6.2) = F(4) = 5$ 。差值  $Distance\_2 = F(A.3) = F(6.3) = F(3) = 3$ 。

步骤 1:

$59 > Data[8.1] = Data[7] = 46$ , 表示数据在第 8 笔数据之后。

$Root = Root + Distance\_2 = 8 + 3 = 11$

$Distance\_1 = Distance\_1 - Distance\_2 = 5 - 3 = 2$

$Distance\_2 = Distance\_2 - Distance\_1 = 3 - 2 = 1$

步骤 2:

$59 < Data[11.1] = Data[10] = 92$ , 表示数据在第 11 笔数据之前。

$Root = Root - Distance\_2 = 11 - 1 = 10$

$Temp = Distance\_1 = 2$

$Distance\_1 = Distance\_2 = 1$

$Distance\_2 = Temp - Distance\_2 = 2 - 1 = 1$

## 步骤 3:

$59 < \text{Data}[10.1] = \text{Data}[9] = 80$ , 表示数据在第 10 笔数据之前。

$\text{Root} = \text{Root} - \text{Distance\_2} = 10 - 1 = 9$

$\text{Temp} = \text{Distance\_1} = 1$

$\text{Distance\_1} = \text{Distance\_2} = 1$

$\text{Distance\_2} = \text{Temp} - \text{Distance\_2} = 1 - 1 = 0$

## 步骤 4:

$59 > \text{Data}[9.1] = \text{Data}[8] = 58$ , 表示数据在第 9 笔数据之后。

$\text{Root} = \text{Root} + \text{Distance\_2} = 9 + 0 = 9$

$\text{Distance\_1} = \text{Distance\_1} - \text{Distance\_2} = 1 - 0 = 1$

$\text{Distance\_2} = \text{Distance\_1} - \text{Distance\_2} = 1 - 1 = 0$

但此时  $\text{Distance\_2}$  为负, 表示未能查找到数据。

## 程序实例:

设计费氏查找法的程序。

## 程序构思:

假设数据有  $n$  笔, 则  $F(a) \leq n + 1$ , 取得  $a$  之后,  $\text{Root} = F(A.1)$ ,  $\text{Distance\_1} = F(A.2)$ ,  $\text{Distance\_2} = F(A.3)$

如果  $\text{KeyValue} < \text{Data}[\text{Root}-1]$ :

表示  $\text{KeyValue}$  可能出现在  $\text{Data}[\text{Root}-1]$  之前。

此时  $\text{Root} = \text{Root} - \text{Distance\_2}$

$\text{Temp} = \text{Distance\_1}$

$\text{Distance\_1} = \text{Distance\_2}$

$\text{Distance\_2} = \text{Temp} - \text{Distance\_2}$

如果  $\text{KeyValue} > \text{Data}[\text{middle}]$ :

表示  $\text{KeyValue}$  可能出现在  $\text{Data}[\text{Root}+1]$  之后。

此时  $\text{Root} = \text{Root} + \text{Distance\_2}$

$\text{Distance\_1} = \text{Distance\_1} - \text{Distance\_2}$

$\text{Distance\_2} = \text{Distance\_2} - \text{Distance\_1}$

如果  $\text{KeyValue} = \text{Data}[\text{middle}]$ :

表示已查找到数据。

重复执行上述 3 个步骤直到找到欲查找数据或  $\text{Distance\_2}$  为负为止。

## 程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: f_search.c */
03 /* 程序目的: 设计费氏查找法的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #define Max 20
07 int Data[Max] = { 12, 16, 19, 22, 25,
```

```

08 32, 39, 48, 55, 57,
09 58, 63, 68, 69, 70,
10 78, 84, 88, 90, 97}; /* 数据数组 */
11 int Counter = 1; /* 计数器 */
12
13 /* ----- */
14 /* 递归求费氏级数 */
15 /* ----- */
16 int Fib(int N)
17 {
18 if (N <= 1) /* 递归结束条件 */
19 return N;
20 else
21 return Fib(N-1) + Fib(N-2); /* 递归执行部分 */
22 }
23
24 /* ----- */
25 /* 费氏查找法 */
26 /* ----- */
27 int Fibonacci_Search(int n,int Key)
28 {
29 int Root; /* 左边界变量 */
30 int Distance_1; /* 上一个费氏数 */
31 int Distance_2; /* 上二个费氏数(差值) */
32 int Temp;
33
34 Root = Fib(n-1);
35 Distance_1 = Fib(n-2);
36 Distance_2 = Fib(n-3);
37
38 Do
39 {
40 if (Key < Data[Root-1]) /* 欲查找值较小 */
41 {
42 /* 查找前半段 */
43 Root = Root - Distance_2;
44 Temp = Distance_1;
45 Distance_1 = Distance_2;
46 Distance_2 = Temp - Distance_2;
47 }
48 else if (Key > Data[Root-1]) /* 欲查找值较大 */
49 {
50 /* 查找后半段 */
51 Root = Root + Distance_2;
52 Distance_1 = Distance_1 - Distance_2;
53 Distance_2 = Distance_2 - Distance_1;
54 }
55 else if (Key == Data[Root-1]) /* 查找到数据 */
56 {
57 printf ("Data[%d] = %d\n",Root-1,Data[Root-1]);
58 return 1;
59 }
60 Counter++;
61 } while (Distance_2 >= 0);
62 return 0;
63 }
64
65 /* ----- */
66 /* 主程序 */
67 /* ----- */
68 void main ()
69 {
70 int KeyValue; /* 欲查找数据变量 */

```

```

69 int FinA;
70
71 printf("Please enter your key value : ");
72 scanf("%d",&KeyValue);
73
74 FinA = 1;
75 while (Fib(FinA) <= Max)
76 FinA++;
77
78 if (Fibonacci_Search(FinA,KeyValue))
79 printf("Search Time = %d\n",Counter); /* 输出查找次数 */
80 else
81 printf("No Found!!\n"); /* 输出没有找到数据 */
82 }

```

运行结果:

```

C:\DS>f_search
Please enter your key value : 12
Data[0] = 12
Search Time = 6

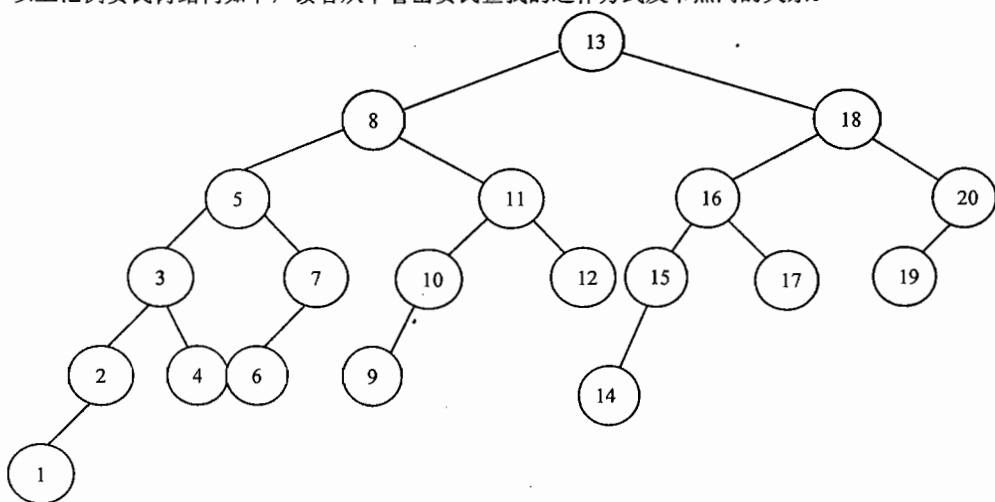
C:\DS>f_search
Please enter your key value : 63
Data[11] = 63
Search Time = 4

C:\DS>f_search
Please enter your key value : 68
Data[12] = 68
Search Time = 1

C:\DS>

```

以上范例费氏树结构如下, 读者从中看出费氏查找的运作方式及节点间的关系。



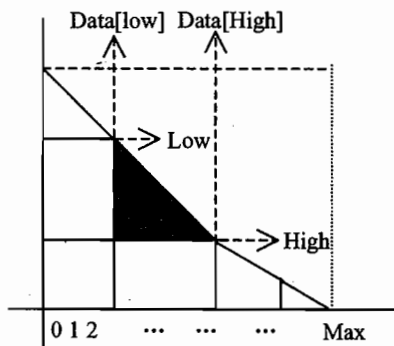
## 9.5 插补查找

对于以上各节的查找方法,其原理都是将数据的范围缩小后,再进行数据的查找,比如在线性查找中,是先查找最前面的数据,如果未能查找到相对的数据,则往下一笔查找,此时数据的范围从原本的  $N$  笔数据,改变为  $N-1$  笔数据,直到查找到数据或查找到最后一笔数据为止。折半查找法,则是将数据的范围,一次次的对分,直到找到数据或只数据范围为 0 为止。而费氏查找法,则是将数据的范围依费氏树的结构来往下查找数据,直到找到数据或已达到费氏树的底端为止。

这一节我们所要学习的是插补查找(Interpolation Searching)。插补查找是一种类似折半查找的查找方法,插补查找所采用的是“各个击破法(Divide-and-Conquer)”的方式来查找数据的位置,只不过与欲查找数据比较的不是数据的中间项,而是以内插法按照比例所选出来的一项。

其方式是以比例的概念,求出欲查找数据可能的位置,然后进行比较,如果该值比欲查找值小,表示欲查找值可能出现在该值之前的范围,如果该值比欲查找值大,则表示欲查找值可能出现在该值之后的范围,一直缩小查找的数据范围,直到找到欲查找数据或数据范围为零为止。

假设数据分布图如下,Low 为数据范围的左边边界(即下限)、High 为数据范围的右边边界(即上限)、KeyValue 为欲查找值,该值的位置介于 Low 和 High 范围间、Data[Low]为左边边界的数据值、Data[High]为右边边界的数据值。Middle 为与欲查找值做比较工作的数据位置。



$$\text{其公式为 } \text{Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

首先判断如果求出来的 Middle 值小于 Low, 则 Middle 值等于 Low; 如果求出来的 Middle 值大于 High, 则 Middle 值等于 High。接下来则按下列规则运作:

1. 若欲查找值小于 Data[Middle]:  
表示数据在 Data[Middle]之前, 则我们必须查找 Data[Middle]之前的数据, 数据范围的右边边界(即上限)High 为 Middle - 1。
2. 若欲查找值大于 Data[Middle]:  
表示数据在 Data[Middle]之后, 则我们必须查找 Data[Middle]之后的数据, 数据范围的左边边界(即下限)Low 为 Middle + 1。
3. 若左边边界(即下限)Low 小于数据范围的右边边界(即上限)High:  
表示数据未查找完成, 重新产生新的 Middle 值。

$$\text{Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

否则,  $\text{Middle} = \text{Low}$ 。

4. 若  $\text{Middle}$  小于左边界(即上限) $\text{Low}$ :

则  $\text{Middle} = \text{Low}$ 。

5. 若  $\text{Middle}$  大于右边界(即下限) $\text{High}$ :

则  $\text{Middle} = \text{High}$ 。

重复执行上述 5 个步骤直到找到欲查找数据或  $\text{Low}$  值大于  $\text{High}$  值为止。

例如: 有一个整数数组的数据内容如下:

|      |   |   |   |   |   |    |    |    |    |    |    |    |
|------|---|---|---|---|---|----|----|----|----|----|----|----|
|      | 0 | 1 | 2 | 3 | 4 | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| Data | 1 | 5 | 6 | 7 | 9 | 12 | 15 | 18 | 27 | 29 | 31 | 33 |

如果我们现在想找出 9, 此时数据 12 笔, 则此时数据范围的左边界( $\text{Low}$ )为 0、数据范围的右边界( $\text{High}$ )为 11、欲查找值( $\text{KeyValue}$ )为 9、左边边界的数据值( $\text{Data}[\text{Low}]$ )为 1、右边边界的数据值( $\text{Data}[\text{High}]$ )为 33。

$$\text{所以 Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

$$= 0 + \frac{(9 - \text{Data}[0]) * (11 - 0)}{\text{Data}[11] - \text{Data}[0]} = 0 + \frac{(9 - 1) * (11 - 0)}{33 - 1}$$

$$= 0 + \frac{88}{32} = 2$$

|      |   |   |   |   |   |    |    |    |    |    |    |    |
|------|---|---|---|---|---|----|----|----|----|----|----|----|
|      | 0 | 1 | 2 | 3 | 4 | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| Data | 1 | 5 | 6 | 7 | 9 | 12 | 15 | 18 | 27 | 29 | 31 | 33 |

步骤 1:

$\text{KeyValue} = 9 > \text{Data}[\text{Middle}] = \text{Data}[2] = 6$ , 表示在第 3 笔数据之后。

所以  $\text{Low} = \text{Middle} + 1 = 2 + 1 = 3$

$$\text{新的 Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

$$= 3 + \frac{(9 - \text{Data}[3]) * (11 - 3)}{\text{Data}[11] - \text{Data}[3]} = 3 + \frac{(9 - 7) * (11 - 3)}{33 - 7}$$

$$= 3 + \frac{2 * 8}{26} \approx 3$$

步骤 2:

$\text{KeyValue} = 9 > \text{Data}[\text{Middle}] = \text{Data}[3] = 7$ , 表示在第 4 笔数据之后。

所以  $\text{Low} = \text{Middle} + 1 = 3 + 1 = 4$

$$\text{新的 Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

$$= 4 + \frac{(9 - \text{Data}[4]) * (11 - 4)}{\text{Data}[11] - \text{Data}[4]} = 4 + \frac{(9 - 9) * (11 - 4)}{33 - 9}$$

$$= 4 + \frac{0 * 7}{24} = 4$$

步骤 3:

$$\text{KeyValue} = 9 = \text{Data}[\text{Middle}] = \text{Data}[4] = 9$$

表示找到数据。

如果我们现在想找出 20, 此时数据 12 笔, 则此时数据范围的左边界(Low)为 0、数据范围的右边界(High)为 11、欲查找值(KeyValue)为 20、左边边界的数据值(Data[Low])为 1、右边边界的数据值(Data[High])为 33。

$$\text{所以 Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

$$= 0 + \frac{(20 - \text{Data}[0]) * (11 - 0)}{\text{Data}[11] - \text{Data}[0]} = 0 + \frac{(20 - 1) * (11 - 0)}{33 - 1}$$

$$= 0 + \frac{209}{32} = 6$$

步骤 1:

$$\text{KeyValue} = 20 > \text{Data}[\text{Middle}] = \text{Data}[6] = 15, \text{表示在第 7 笔数据之后。}$$

$$\text{所以 Low} = \text{Middle} + 1 = 6 + 1 = 7$$

$$\text{新的 Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

$$= 7 + \frac{(20 - \text{Data}[7]) * (11 - 7)}{\text{Data}[11] - \text{Data}[7]} = 7 + \frac{(20 - 18) * (11 - 7)}{33 - 18}$$

$$= 7 + \frac{8}{15} \approx 7$$

步骤 2:

$$\text{KeyValue} = 20 > \text{Data}[\text{Middle}] = \text{Data}[7] = 18, \text{表示在第 8 笔数据之后。}$$

$$\text{所以 Low} = \text{Middle} + 1 = 7 + 1 = 8$$

$$\text{新的 Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

$$= 8 + \frac{(20 - \text{Data}[8]) * (11 - 8)}{\text{Data}[11] - \text{Data}[8]} = 8 + \frac{(20 - 27) * (11 - 8)}{33 - 27}$$

$$= 8 + \frac{(-7) * 3}{6} = 5$$

因为此时  $\text{Middle} = 5 < \text{Low} = 8$ , 所以  $\text{Middle} = \text{Low} = 8$ 。

步骤 3:

$\text{KeyValue} = 20 < \text{Data}[\text{Middle}] = \text{Data}[8] = 27$ , 表示在第 9 笔数据之前。

所以  $\text{High} = \text{Middle} - 1 = 8 - 1 = 7$

因为此时  $\text{High} = 7 < \text{Low} = 8$ , 所表示未能查找到数据。

插补查找法对于平均分布的数据效率很高, 其时间复杂度为  $O(\log_2 \log_2 n)$ , 比折半查找法还要好, 不过对于不均匀分布的数据效率却不好, 其最坏状况时的时间复杂度可能达到  $O(n)$ 。

程序实例:

设计插补查找法的程序。

程序构思:

假设数据有  $n$  笔, 则此时欲查找为  $\text{KeyValue}$ 。

数据范围的左边界( $\text{Low}$ )为 0

数据范围的右边界( $\text{High}$ )为  $n-1$ 、

左边界的数据值为  $\text{Data}[\text{Low}]$

右边界的数据值为  $\text{Data}[\text{High}]$

欲比较数据的位置为

$$\text{Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

如果  $\text{KeyValue}$  小于  $\text{Data}[\text{Middle}]$ :

表示数据在  $\text{Data}[\text{Middle}]$  之前,

则  $\text{High} = \text{Middle} - 1$ 。

如果  $\text{KeyValue}$  大于  $\text{Data}[\text{Middle}]$ :

表示数据在  $\text{Data}[\text{Middle}]$  之后

则  $\text{Low} = \text{Middle} + 1$ 。

如果  $\text{Low}$  小于  $\text{High}$ :

重新产生新的  $\text{Middle}$  值。

$$\text{Middle} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$$

否则,  $\text{Middle} = \text{Low}$ 。

如果  $\text{Middle}$  小于  $\text{Low}$ :

则  $\text{Middle} = \text{Low}$ 。

如果  $\text{Middle}$  大于  $\text{High}$ :

则  $\text{Middle} = \text{High}$ 。

重复执行上述 5 个步骤直到找到欲查找数据或  $\text{Low}$  大于  $\text{High}$  为止。

程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: i_search.c */
03 /* 程序目的: 设计插补查找法的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #define Max 20
```



```

07 int Data[Max] = { 12, 16, 19, 22, 25,
08 32, 39, 48, 55, 57,
09 58, 63, 68, 69, 70,
10 78, 84, 88, 90, 97}; /* 数据数组 */
11 int Counter = 1; /* 计数器 */
12
13 /* ----- */
14 /* 插补查找法 */
15 /* ----- */
16 int Interpolation_Search(int Key)
17 {
18 int Low; /* 插补查找法左边界变量 */
19 int High; /* 插补查找法右边界变量 */
20 int Middle; /* 插补查找法中间数 */
21
22 Low = 0;
23 High = Max - 1;
24 Middle = Low + (Key - Data[Low]) * (High - Low)
25 / (Data[High] - Data[Low]);
26
27 if (Middle < Low)
28 Middle = Low;
29 if (Middle > High)
30 Middle = High;
31
32 while (Low <= High)
33 {
34 if (Key < Data[Middle]) /* 欲查找值较小 */
35 High = Middle - 1; /* 查找前半段 */
36 else if (Key > Data[Middle]) /* 欲查找值较大 */
37 Low = Middle + 1; /* 查找后半段 */
38
39 else if (Key == Data[Middle]) /* 查找到数据 */
40 {
41 printf ("Data[%d] = %d\n",Middle,Data[Middle]);
42 return 1;
43 }
44
45 if (Low < High)
46 Middle = Low + (Key - Data[Low]) * (High - Low)
47 / (Data[High] - Data[Low]);
48 if (Middle < Low)
49 Middle = Low;
50 if (Middle > High)
51 Middle = High;
52
53 Counter++;
54 }
55 return 0;
56 }
57
58 /* ----- */
59 /* 主程序 */
60 /* ----- */
61 void main ()
62 {
63 int KeyValue; /* 欲查找数据变量 */
64
65 printf("Please enter your key value : ");
66 scanf ("%d",&KeyValue);
67 if (Interpolation_Search(KeyValue))

```

```

68 printf("Search Time = %d\n",Counter); /* 输出查找次数
69 */
70 else
71 printf("No Found!!\n"); /* 输出没有找到数据 */
72 }

```

运行结果:

```

C:\DS>i_search
Please enter your key value : 55
Data[8] = 55
Search Time = 2

C:\DS>i_search
Please enter your key value : 99
No Found!!

C:\DS>

```

因为上述的程序中,数据是平均分布,每一笔数据值差距并不大,所以上述程序如果查找已存在的数据最多也只要3次就可找到该数据。但是如果数据是不平均分布的,可能会浪费多次的查找时间。

我们将原数据内容换成下列的数据,重新运行程序之后,我们会发现,查找的次数变多了。

```

int Data[20] = { 12, 160, 219, 522, 725,
 732, 739, 748, 755, 757,
 958, 963, 1068, 1169, 1570
 2278, 5384, 8888, 9000, 9997}; /* 数据数组 */

```

数据替换后,程序运行结果如下:

```

C:\DS>i_search
Please enter your key value : 755
Data[8] = 755
Search Time = 8

C:\DS>i_search
Please enter your key value : 522
Data[3] = 522
Search Time = 4

C:\DS>i_search
Please enter your key value : 1570
Data[14] = 1570
Search Time = 10

C:\DS>i_search
Please enter your key value : 68
No Found!!

C:\tc>

```

所以对于不平均分布的数据,采用插补查找法,不但无法增快查找速度,反而让最差状况的时间复杂度达到  $O(n)$ ,与线性查找法相同。

以下我们将介绍一种改良型的插补查找法,我们称为加强型插补查找法(Robust Interpolation

Searching)。

改良后的插补查找法，是取

1.  $\text{Num1} = \text{Low} + \text{Gap}$
2.  $\text{Num2} = \text{High} - \text{Gap}$
3.  $\text{Num3} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$

3 数中的中间数作为 Middle 值，其中  $\text{Gap} = \sqrt{\text{High} - \text{Low} + 1}$ 。

程序实例：

设计加强型的插补查找法程序。

程序构思：

假设数据有  $n$  笔，则此时欲查找为 KeyValue。

数据范围的左边边界(Low)为 0

数据范围的右边边界(High)为  $n-1$ 、

左边边界的数据值为  $\text{Data}[\text{Low}]$

右边边界的数据值为  $\text{Data}[\text{High}]$

$$\text{Gap} = \sqrt{\text{High} - \text{Low} + 1}$$

欲比较数据的位置为 Middle 取下列 3 数的中间数：

1.  $\text{Num1} = \text{Low} + \text{Gap}$
2.  $\text{Num2} = \text{High} - \text{Gap}$
3.  $\text{Num3} = \text{Low} + \frac{(\text{KeyValue} - \text{Data}[\text{Low}]) * (\text{High} - \text{Low})}{\text{Data}[\text{High}] - \text{Data}[\text{Low}]}$

如果 KeyValue 小于  $\text{Data}[\text{Middle}]$ ：

表示数据在  $\text{Data}[\text{Middle}]$  之前，

则  $\text{High} = \text{Middle} - 1$ 。

如果 KeyValue 大于  $\text{Data}[\text{Middle}]$ ：

表示数据在  $\text{Data}[\text{Middle}]$  之后

则  $\text{Low} = \text{Middle} + 1$ 。

如果 Low 小于 High：

重新产生新的 Middle 值。

否则， $\text{Middle} = \text{Low}$ 。

如果 Middle 小于 Low：

则  $\text{Middle} = \text{Low}$ 。

如果 Middle 大于 High：

则  $\text{Middle} = \text{High}$ 。

重复执行上述5个步骤直到找到欲查找数据或 Low 大于 High 为止。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: o_search.c */
03 /* 程序目的: 设计插补查找法的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #define Max 20
07 int Data[Max] = { 12, 160, 219, 522, 725,
08 732, 739, 748, 755, 757,
09 958, 963, 1068, 1169, 1570,
10 2278, 5384, 8888, 9000, 9997}; /* 数据数组 */
11 int Counter = 1; /* 计数器 */
12
13 /* ----- */
14 /* 找出加强型插补查找的中间值 */
15 /* ----- */
16 int FindRobust(int Low,int High,int Key)
17 {
18 int Gap; /* 差值 */
19 int Num1; /* 数值1 */
20 int Num2; /* 数值2 */
21 int Num3; /* 数值3 */
22
23 Gap = ceil (sqrt((float) High - (float) Low + 1.0));
24 Num1 = Low + Gap; /* 计算数值1 */
25 Num2 = High - Gap; /* 计算数值2 */
26 Num3 = Low + (Key - Data[Low]) * (High - Low)
27 / (Data[High] - Data[Low]); /* 计算数值3 */
28
29 if (Num1 >= Num2)
30 if (Num2 >= Num3)
31 return Num2; /* Num1 >= Num2 >= Num3 */
32 else if (Num1 >= Num3)
33 return Num3; /* Num1 >= Num3 >= Num2 */
34 else
35 return Num1; /* Num3 >= Num1 >= Num2 */
36 else if (Num2 >= Num1)
37 if (Num1 >= Num3)
38 return Num1; /* Num2 >= Num1 >= Num3 */
39 else if (Num3 >= Num1)
40 return Num3; /* Num2 >= Num3 >= Num1 */
41 else
42 return Num2; /* Num3 >= Num2 >= Num1 */
43 return 0;
44 }
45
46 /* ----- */
47 /* 插补查找法 */
48 /* ----- */
49 int Interpolation_Search(int Key)
50 {
51 int Low; /* 插补查找法左边界变量 */
52 int High; /* 插补查找法右边界变量 */
53 int Middle; /* 插补查找法中间数 */

```

```

54
55 Low = 0;
56 High = Max - 1;
57 Middle = FindRobust(Low, High, Key);
58
59 if (Middle < Low)
60 Middle = Low;
61 if (Middle > High)
62 Middle = High;
63
64 while (Low <= High)
65 {
66 if (Key < Data[Middle]) /* 欲查找值较小 */
67 High = Middle - 1; /* 查找前半段 */
68 else if (Key > Data[Middle]) /* 欲查找值较大 */
69 Low = Middle + 1; /* 查找后半段 */
70
71 else if (Key == Data[Middle]) /* 查找找到数据 */
72 {
73 printf ("Data[%d] = %d\n", Middle, Data[Middle]);
74 return 1;
75 }
76 if (Low < High)
77 Middle = FindRobust(Low, High, Key);
78 if (Middle < Low)
79 Middle = Low;
80 if (Middle > High)
81 Middle = High;
82 Counter++;
83 }
84 return 0;
85 }
86
87 /* ----- */
88 /* 主程序 */
89 /* ----- */
90 void main ()
91 {
92 int KeyValue; /* 欲查找数据变量 */
93
94 printf("Please enter your key value : ");
95 scanf("%d", &KeyValue);
96
97 if (Interpolation_Search(KeyValue))
98 printf("Search Time = %d\n", Counter); /* 输出查找次数 */
99 else
100 printf("No Found!!\n"); /* 输出没有找到数据 */
101 }

```

运行结果:

```

C:\DS>o_search
Please enter your key value : 755
Data[8] = 755
Search Time = 8

C:\DS>o_search
Please enter your key value : 522
Data[3] = 522
Search Time = 4

```

```

C:\DS>o_search
Please enter your key value : 1570
Data[14] = 1570
Search Time = 10

C:\DS>o_search
Please enter your key value : 68
No Found!!

C:\DS>

```

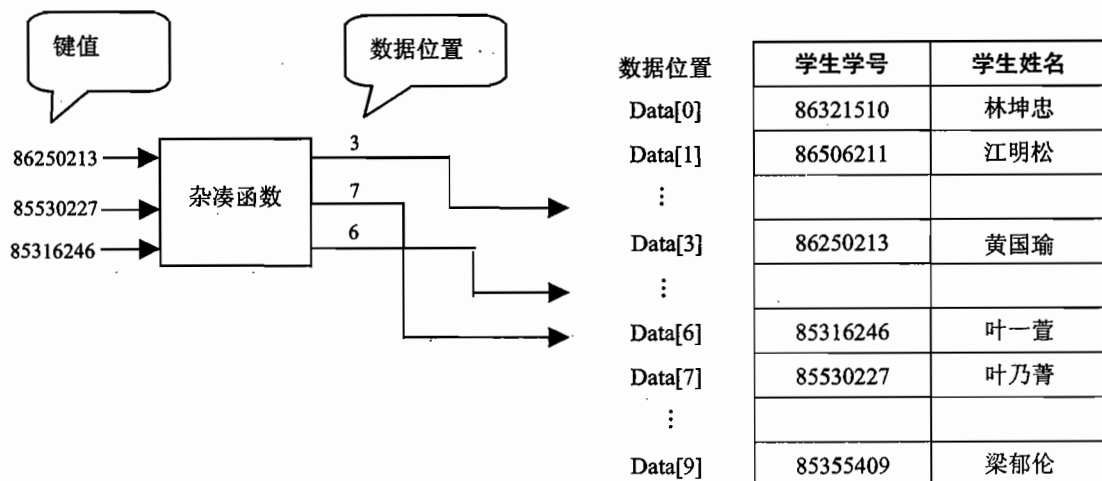
加强型的插补查找法, 对于平均分布的数据, 其时间复杂度为  $O(\log \log n)$ , 而对不均匀分布的数据时间复杂度则为  $O((\log n)^2)$ , 虽然没有折半查找法好, 但是对于改善原来插补查找法最差状况的时间复杂度  $O(n)$ , 却有明显的差别。

## 9.6 杂凑查找

前面我们谈了那么多的查找的方法, 每一个查找法都需要经过一次又一次的比较才能找到欲查找数据或判断欲查找数据存在不存在。如果想要让查找的速度增快, 减少数据比较的次数是唯一的方法。而杂凑查找(Hash Searching)就是一种可以让数据的比较次数减少到每次查找只需一次即能找出数据的方法。

因为杂凑表中数据存储的位置是通过特定的杂凑函数运算而来的。所有的数据都是通过杂凑函数运算后存储于杂凑表中, 当我们进行杂凑查找时, 只需将数据再通过杂凑函数的计算后, 便可求得数据的位置, 即可进行一次的数据的比较即找到欲查找数据。

在杂凑结构中, 我们称输入数据的值为键值(Key)。例如: 一个利用杂凑表存储的学生数据, 其中学生的学号即为这个杂凑表的键值, 如果存储的内容是学生学号和 student 姓名, 其运作过程大致如下:



### 9.6.1 杂凑函数

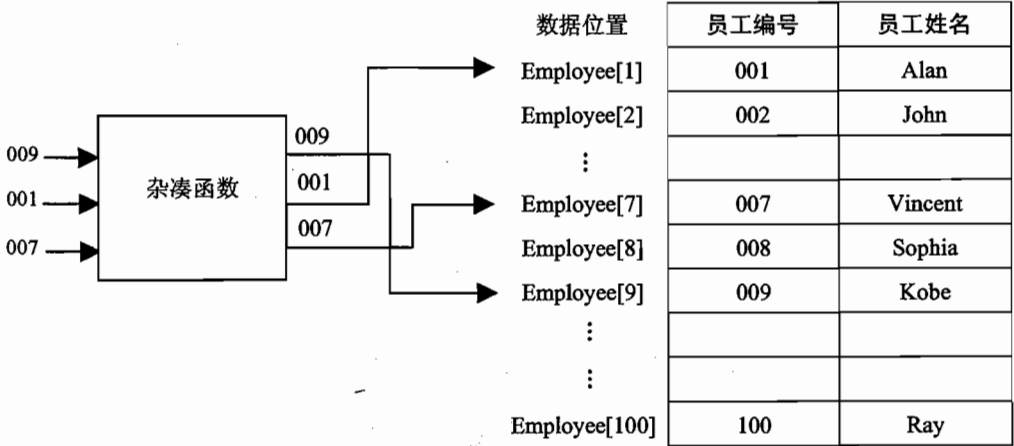
用于处理杂凑表的杂凑函数, 常见的有下列几种:

1. 直接法(Direct method)

直接法就是每一个键值对应一个存储空间，而不经任何的数学运算动作。例如：公司中有一百个员工，而员工编号介于 1 到 100，直接法就是员工的编号即为数据的位置，编号 1 员工的数据在数据中的第一笔、编号 2 员工的数据在数据中的第二笔……，依此类推。

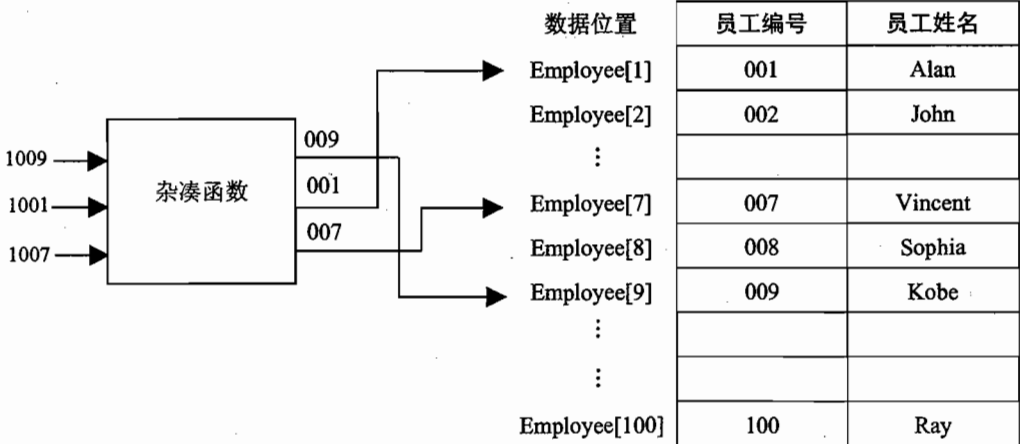
直接法适用于键值较小的数据。如果上一个例子，我们换以身份证字号当键值，因为身份证字号上第 1 个字为英文字、第 2 到 10 个字为数字，则我们至少要预留 99999999 个存储空间，但是我们却只用到其中的 100 个存储空间，造成浪费大量未用的存储空间。

杂凑法大部分是用在将数值范围较大的键值，对应于少数的存储空间，以增加查找的速度，减少存储空间的浪费。



2. 减去法(Subtraction method)

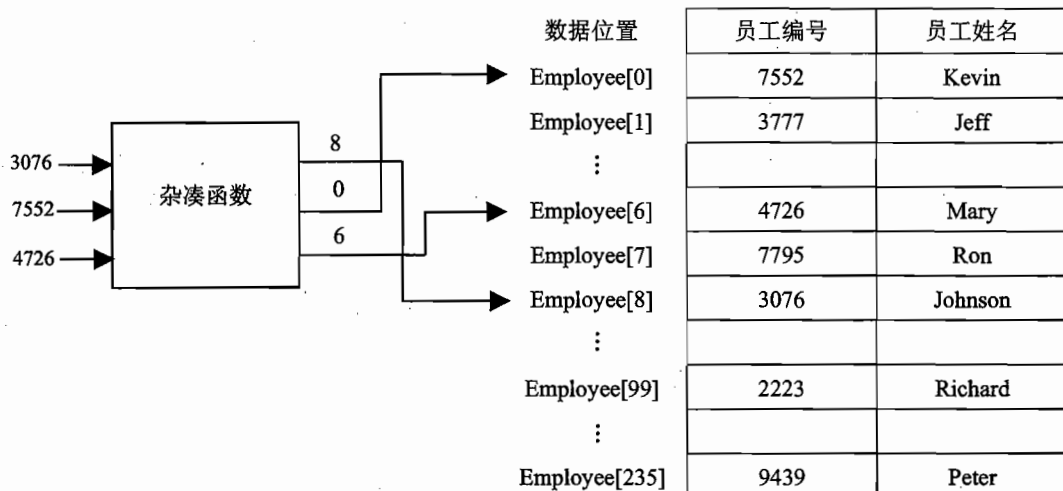
减去法就是数据的键值减去一个特定的数值，以求得数据存储的位置。例如：公司中有 100 个员工，而员工编号介于 1001 到 1100，减去法就是员工的编号减去 1000 后即为员工的数据位置，编号 1001 员工的数据在数据中的第 1 笔、编号 1002 员工的数据在数据中的第 2 笔…，依此类推。因为编号 1000 以前的并没有数据，所有的员工编号皆从 1001 开始编号。



3. 余数法(Modulo-Division method)

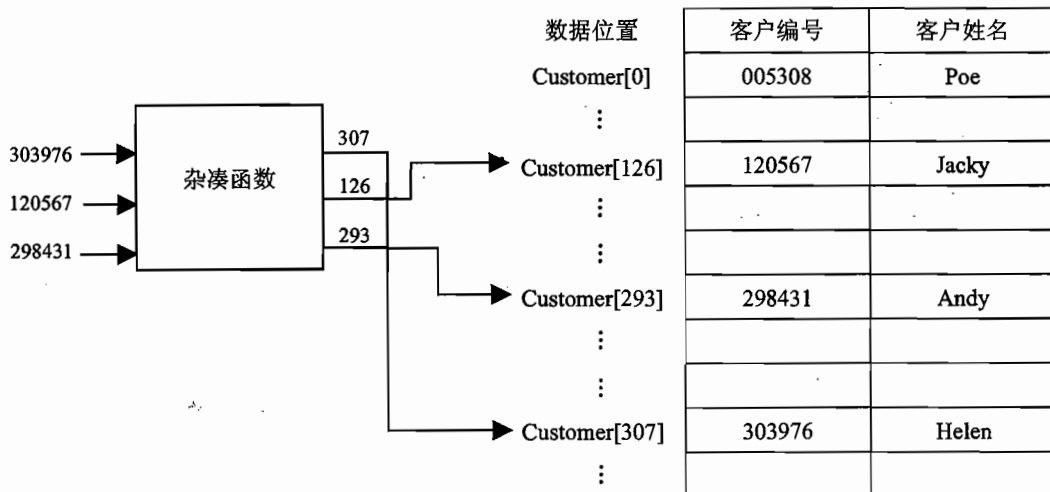
余数法就是将数据的键值除于数组的大小后，取其余数作为数据存储的位置。例如：公司中有 206 个员工，而员工编号介于 1000 到 9999，余数法就是员工的编号除以数据个数减去 236 后，取余数即为

数据的位置, 编号 5428 员工的数据(编号 5428 除以 236 取余数得 0)放在数据中的第一笔、编号 3512 员工的数据(编号 3512 除以 236 取余数得 8)放在数据中的第 9 笔……, 依此类推。



#### 4. 数值抽出法(Digit-Extraction method)

数值抽出法就是将数据的键值中的某几位数取出后作为数据存储的位置。例如, 数据的键值为一组 6 位数的号码。我们假设键值中的第 1 位、第 2 位、第 5 位数作为数据位置的索引值。则:



136781   =>  138  
 215484   =>  218  
 561575   =>  567  
 021157   =>  025

#### 5. 中间平方法(Midsquare method)

中间平方法就是将数据的键值中的前几位数取出后平方产生一个新的数值, 再从新产生的数值中取出中间某几位数作为数据存储的位置。例如, 决定以数据的键值的前 3 位数, 平方后产生的一个新的数值, 再从新的数值中取第二位到第 4 位数作为数据位置的索引值。则:



$325483 \Rightarrow 325 * 325 = 105625 \Rightarrow 056$   
 $213457 \Rightarrow 213 * 213 = 045369 \Rightarrow 453$   
 $654875 \Rightarrow 654 * 654 = 427716 \Rightarrow 277$   
 $215875 \Rightarrow 215 * 215 = 046225 \Rightarrow 462$

#### 6. 折迭法(Folding method)

折迭法就是将数据的键值中分为多层, 然后相加后取其结果作为数据存储的位置。常用的折迭法有两种, 一为各层的数据直接相加, 取其结果, 称为移位折迭法(Fold Shift)。如键值为 123456789, 将键值分为 3 层后, 运算后的结果如下:

$$\begin{array}{r}
 123 \\
 456 \\
 + 789 \\
 \hline
 1368
 \end{array}$$

即取 368 作为数据存储的位置。

另一种为各层的数据在相加之前每隔一部分便进行一次反转, 相加后取其结果, 称为边界折迭法(Fold Boundary)。如键值为 123456789, 将键值分为 3 层后, 运算后的结果如下:

$$\begin{array}{r}
 321(\text{反转}) \\
 456 \\
 + 987(\text{反转}) \\
 \hline
 1764
 \end{array}$$

即取 764 作为数据存储的位置。

#### 7. 旋转法(Rotation method)

旋转法是将数据的键值中进行旋转。旋转法通常并不直接使用在杂凑函数上, 而是搭配着其它杂凑函数使用。例如有一组 6 位数的键值, 我们将最后一位数, 旋转后放置在第一位数, 其余的往右移。

| 原键值     | 旋转过程                | 旋转后的新键值             |
|---------|---------------------|---------------------|
| 5062101 | 506210 <del>1</del> | <del>1</del> 506210 |
| 5062102 | 506210 <del>2</del> | <del>2</del> 506210 |
| 5062103 | 506210 <del>3</del> | <del>3</del> 506210 |
| 5062104 | 506210 <del>4</del> | <del>4</del> 506210 |
| 5062105 | 506210 <del>5</del> | <del>5</del> 506210 |



#### 8. 伪随机数法(Pseudo-random method)

伪随机数法是将利用数据的键值经过伪随机数法的运算后的结果作为数据存储的位置。其公式如下(a 和 c 为质数):

$$Y = (a * \text{Key} + c) \text{ modulo 数组的大小}$$

如现在的键值为 321547，数组大小为 107，我们取  $a=13$ 、 $c=5$  则：

$$\begin{aligned} Y &= (13 * 321547 + 5) \% 107 \\ &= (4180111 + 5) \% 107 \\ &= 4180116 \% 107 \\ &= 54 \end{aligned}$$

即取 54 作为数据存储的位置。

## 9.6.2 杂凑碰撞解决法

当然利用杂凑函数所产生出的数据位置可能会发生数据位置已存在有一笔数据的情形，此时我们称为杂凑碰撞(Hash Collision)。发生杂凑碰撞时，我们必须提供一些解决方法，才能让数据有对映的存储位置，否则会造成数据流失的错误。

常见的杂凑碰撞解决法有下列几种：

### 1. 线性开放寻址法(Linear Open Addressing)

而线性开放寻址法包括了：











#### ● 线性探索法(Linear Probe)：

线性探索法所采用的就是当杂凑函数产生的数据地址已有数据存在时，即发生杂凑碰撞。处理的原则是往下一笔数据位置寻找可用空间存储数据。










假设现在有数据如下：

654 638 214 357 376 854 652 392









现在我们采用数值抽出法，取得第二位数，并将数据放入大小为 10 个杂凑表中，一开始杂凑表中皆无数据。

|      |                                                                                     |                                                                                     |                                                                                     |                                                                                     |                                                                                     |                                                                                     |                                                                                     |                                                                                     |                                                                                     |                                                                                      |
|------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
|      | 0                                                                                   | 1                                                                                   | 2                                                                                   | 3                                                                                   | 4                                                                                   | 5                                                                                   | 6                                                                                   | 7                                                                                   | 8                                                                                   | 9                                                                                    |
| Data |  |  |  |  |  |  |  |  |  |  |

插入数据 654，位置为 5：

|      |                                                                                     |                                                                                     |                                                                                     |                                                                                     |                                                                                     |     |                                                                                     |                                                                                     |                                                                                     |                                                                                      |
|------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-----|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
|      | 0                                                                                   | 1                                                                                   | 2                                                                                   | 3                                                                                   | 4                                                                                   | 5   | 6                                                                                   | 7                                                                                   | 8                                                                                   | 9                                                                                    |
| Data |  |  |  |  |  | 654 |  |  |  |  |

插入数据 638，位置为 3：

|      |                                                                                     |                                                                                     |                                                                                     |     |                                                                                     |     |                                                                                     |                                                                                     |                                                                                     |                                                                                      |
|------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-----|-------------------------------------------------------------------------------------|-----|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
|      | 0                                                                                   | 1                                                                                   | 2                                                                                   | 3   | 4                                                                                   | 5   | 6                                                                                   | 7                                                                                   | 8                                                                                   | 9                                                                                    |
| Data |  |  |  | 638 |  | 654 |  |  |  |  |

插入数据 214，位置为 1：

|      |   |     |   |     |   |     |   |   |   |   |
|------|---|-----|---|-----|---|-----|---|---|---|---|
|      | 0 | 1   | 2 | 3   | 4 | 5   | 6 | 7 | 8 | 9 |
| Data |   | 214 |   | 638 |   | 654 |   |   |   |   |

插入数据 357, 位置为 5, 发生杂凑碰撞, 往下一笔数据位置寻找可用空间存储数据, 位置为 6:

|      |   |     |   |     |   |     |     |   |   |   |
|------|---|-----|---|-----|---|-----|-----|---|---|---|
|      | 0 | 1   | 2 | 3   | 4 | 5   | 6   | 7 | 8 | 9 |
| Data |   | 214 |   | 638 |   | 654 | 357 |   |   |   |

插入数据 376, 位置为 7:

|      |   |     |   |     |   |     |     |     |   |   |
|------|---|-----|---|-----|---|-----|-----|-----|---|---|
|      | 0 | 1   | 2 | 3   | 4 | 5   | 6   | 7   | 8 | 9 |
| Data |   | 214 |   | 638 |   | 654 | 357 | 376 |   |   |

插入数据 854, 位置为 5, 发生杂凑碰撞, 往下一笔数据位置寻找可用空间存储数据, 位置为 6, 再次发生杂凑碰撞, 再往下一笔数据位置寻找可用空间存储数据, 位置为 7, 再次发生杂凑碰撞, 再往下一笔数据位置寻找可用空间存储数据, 位置为 8:

|      |   |     |   |     |   |     |     |     |     |   |
|------|---|-----|---|-----|---|-----|-----|-----|-----|---|
|      | 0 | 1   | 2 | 3   | 4 | 5   | 6   | 7   | 8   | 9 |
| Data |   | 214 |   | 638 |   | 654 | 357 | 376 | 854 |   |

插入数据 662, 位置为 6, 发生杂凑碰撞, 往下一笔数据位置寻找可用空间存储数据, 位置为 7, 再次发生杂凑碰撞, 再往下一笔数据位置寻找可用空间存储数据, 位置为 8, 再次发生杂凑碰撞, 再往下一笔数据位置寻找可用空间存储数据, 位置为 9:

|      |   |     |   |     |   |     |     |     |     |     |
|------|---|-----|---|-----|---|-----|-----|-----|-----|-----|
|      | 0 | 1   | 2 | 3   | 4 | 5   | 6   | 7   | 8   | 9   |
| Data |   | 214 |   | 638 |   | 654 | 357 | 376 | 854 | 662 |

插入数据 392, 位置为 9, 再次发生杂凑碰撞, 再往下一笔数据位置寻找可用空间存储数据, 位置为 0:

|      |     |     |   |     |   |     |     |     |     |     |
|------|-----|-----|---|-----|---|-----|-----|-----|-----|-----|
|      | 0   | 1   | 2 | 3   | 4 | 5   | 6   | 7   | 8   | 9   |
| Data | 392 | 214 |   | 638 |   | 654 | 357 | 376 | 854 | 662 |

#### ● 二次方探索法(Quadratic Probe)

二次方探索法所采用的就是当杂凑函数产生的数据地址已有数据存在时, 即发生杂凑碰撞。处理的原则以现在的数据地址加上碰撞次数的平方数, 当数据地址超出数组大小时, 则让数据地址采以循环的方式处理(即新数据地址对数组大小取余数)。

假设第一次发生杂凑碰撞的位置在 1, 数组的大小为 80, 则其运作方式如下表所示:

| 探索次数 | 杂凑碰撞位置 | 位置增加值      | 新的数据位置                 |
|------|--------|------------|------------------------|
| 1    | 1      | $1^2 = 1$  | $(1 + 1) \% 80 = 2$    |
| 2    | 2      | $2^2 = 4$  | $(2 + 4) \% 80 = 6$    |
| 3    | 6      | $3^2 = 9$  | $(6 + 9) \% 80 = 15$   |
| 4    | 15     | $4^2 = 16$ | $(15 + 16) \% 80 = 31$ |
| 5    | 31     | $5^2 = 25$ | $(31 + 25) \% 80 = 56$ |
| 6    | 56     | $6^2 = 36$ | $(56 + 36) \% 80 = 12$ |
| 7    | 12     | $7^2 = 49$ | $(12 + 49) \% 80 = 61$ |
| 8    | 61     | $8^2 = 36$ | $(61 + 36) \% 80 = 17$ |

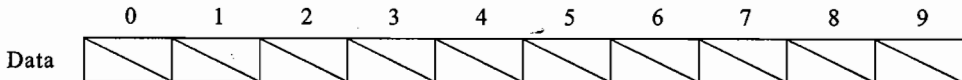
## 2. 差值解决法(Offset Resolution)

差值解决法所采用的就是当杂凑函数产生的数据地址已有数据存在时,即发生杂凑碰撞。处理的原则以现在的数据地址加上一个固定的差值,当数据地址超出数组大小时,则让数据地址采以循环的方式处理(即新数据地址对数组大小取余数)。

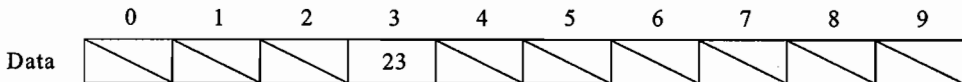
假设现在有数据如下:

23 57 65 63 67 33

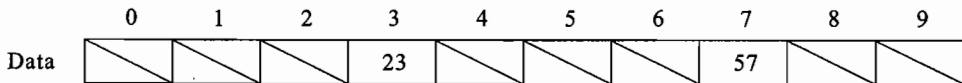
现在我们采用余数法,并将数据放入大小为 10 个杂凑表中,一开始杂凑表中皆无数据,令差值为 3。



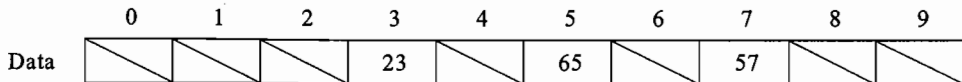
插入数据 23,  $23 \% 10 = 3$ , 位置为 3:



插入数据 57,  $57 \% 10 = 7$ , 位置为 7:



插入数据 65,  $65 \% 10 = 5$ , 位置为 5:



插入数据 63,  $63 \% 10 = 3$ , 位置为 3, 发生杂凑碰撞, 往下一笔的数据位置寻找可用空间存储数据, 位置为原数据位置加上差值  $= 3 + 3 = 6$ :

|      |   |   |   |    |   |    |    |    |   |   |
|------|---|---|---|----|---|----|----|----|---|---|
|      | 0 | 1 | 2 | 3  | 4 | 5  | 6  | 7  | 8 | 9 |
| Data |   |   |   | 23 |   | 65 | 63 | 57 |   |   |

插入数据 67,  $67 \% 10 = 7$ , 位置为 7, 发生杂凑碰撞, 往下一笔的数据位置寻找可用空间存储数据, 位置为原数据位置加上差值  $= 7 + 3 = 10 \Rightarrow 10 \% 10 = 0$ ;

|      |    |   |   |    |   |    |    |    |   |   |
|------|----|---|---|----|---|----|----|----|---|---|
|      | 0  | 1 | 2 | 3  | 4 | 5  | 6  | 7  | 8 | 9 |
| Data | 67 |   |   | 23 |   | 65 | 63 | 57 |   |   |

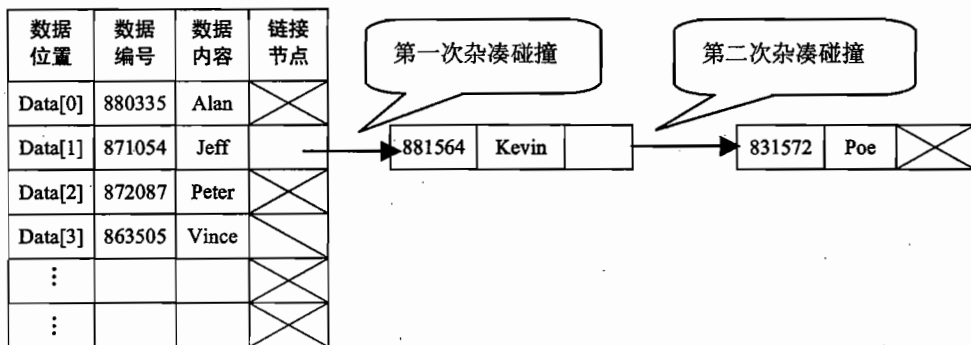
插入数据 33,  $33 \% 10 = 3$ , 位置为 3, 发生杂凑碰撞, 往下一笔的数据位置寻找可用空间存储数据, 位置为原数据位置加上差值  $= 3 + 3 = 6$ , 再发生杂凑碰撞, 往下一笔的数据位置寻找可用空间存储数据, 位置为原数据位置加上差值  $= 6 + 3 = 9$ ;

|      |    |   |   |    |   |    |    |    |   |    |
|------|----|---|---|----|---|----|----|----|---|----|
|      | 0  | 1 | 2 | 3  | 4 | 5  | 6  | 7  | 8 | 9  |
| Data | 67 |   |   | 23 |   | 65 | 63 | 57 |   | 33 |

### 3. 链表解决法(Linked List Resolution)

链表解决法所采用的就是当杂凑函数产生的数据地址已有数据存在时, 即发生杂凑碰撞。处理的原则以现在的数据地址再串连一个新的链表来存储数据。

其运作方式如下图所示:



### 4. 分桶杂凑法(Bucket Hashing)

分桶杂凑法所采用的就是数据分为几个大类, 我们称之为“桶”(Bucket), 而每一个大类中可放置相同大类的数据多笔, 当经杂凑函数运算后属同一大类的数据, 即放在同一大类中, 直到同一大类的数据全填满才往下一大类存储数据, 关于 Bucket 的结构, 我们可用多维的数组来制作。其结构如下图所示:

| 数据位置    | 分桶编号        | 数据编号   | 数据内容  |
|---------|-------------|--------|-------|
| Data[0] | Bucket<br>0 | 880335 | Alan  |
|         |             |        |       |
|         |             |        |       |
| Data[1] | Bucket<br>1 | 871054 | Jeff  |
|         |             | 881564 | Kevin |
|         |             | 831572 | Poe   |
| Data[2] | Bucket<br>2 | 872087 | Peter |
|         |             | 841098 | Bob   |
|         |             |        |       |
| ⋮       |             |        |       |
| Data[9] | Bucket<br>9 | 849035 | Ketty |
|         |             |        |       |
|         |             |        |       |

因 Bucket 1  
已填满!

### 9.6.3 杂凑查找

谈了那么多的杂凑函数与杂凑碰撞解决方法, 现在我们就来试着写成程序。数据必须用杂凑表的结构所存储的, 才能进行杂凑查找。

程序实例:

运用余数法作为杂凑函数及差值解决方法解决杂凑碰撞解决方法、设计杂凑表。

程序构思:

依余数法将数据存储至杂凑表中、发生杂凑碰撞时以差值解决方法解决。

余数法:

$\text{Address} = \text{Key} \% \text{Array\_Size}$

差值解决方法:

$\text{New\_Address} = (\text{Old\_Address} + \text{Offset}) \% \text{Array\_Size}$

数据查找时, 同时也先用余数法运算出数据地址, 若找不到运用差值法找出下一个数据地址, 直到数据全查找完为止。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: h_search.c */
03 /* 程序目的: 运用余数法作为杂凑函数及差值解决方法解决杂 */
04 /* 凑碰撞解决方法、设计杂凑表。 */

```

```

05 /* Written By Kuo-Yu Huang. (WANT Studio.) */
06 /* ===== */
07 #define Max 6
08 #define HashMax 5
09 int Data[Max] = { 12, 160, 219, 522, 725, 9997}; /*
10 数据数组 */
11 int HashTab[HashMax];
12 int Counter = 1; /* 计数器 */
13
14 /* ----- */
15 /* 杂凑函数之余数法 */
16 /* ----- */
17 int Hash_Mod(int Key)
18 {
19
20 return Key % HashMax ; /* 返回 键值除以杂凑表大小取余数 */
21
22 }
23
24 /* ----- */
25 /* 差值杂凑碰撞解决法 */
26 /* ----- */
27 int Collision_Offset(int Address)
28 {
29 int Offset = 3; /* 设差值为 3 */
30
31 return (Address + Offset) % HashMax;
32 /* 返回 旧地址加差值除以杂凑表大小取余数 */
33
34 }
35
36 /* ----- */
37 /* 建立杂凑表 */
38 /* ----- */
39 int Create_Hash(int Key)
40 {
41 int HashTime; /* 杂凑次数 */
42 int CollisionTime; /* 碰撞次数 */
43 int Address; /* 数据地址 */
44 int i;
45
46 HashTime = 0;
47 CollisionTime = 0;
48 Address = Hash_Mod(Key); /* 调用杂凑函数 */
49 while (HashTime < HashMax)
50 {
51 if (HashTab[Address] == 0) /* 可存储数据 */
52 {
53 HashTab[Address] = Key;
54 printf("Key : %d => Address %d\n",Key,Address);
55 for (i=0;i<HashMax;i++)
56 {
57 printf("[%d]",HashTab[i]);
58 }
59 printf("\n");
60 return 1;
61 }
62 else
63 {
64 CollisionTime++;
65 printf("Collision %d => Address

```

```

66 %d\n", CollisionTime, Address);
67 Address = Collision_Offset(Address); /*调用碰撞解决法*/
68 }
69 HashTime++;
70 }
71 return 0;
72 }
73
74 /* ----- */
75 /* 杂凑查找法 */
76 /* ----- */
77 int Hash_Search(int Key)
78 {
79 int Address; /* 数据地址 */
80
81 Counter = 0;
82 Address = Hash_Mod(Key); /* 调用杂凑函数 */
83 while (Counter < HashMax)
84 {
85 Counter++;
86 if (HashTab[Address] == Key)
87 return 1;
88 else
89 Address = Collision_Offset(Address); /*调用碰撞解决法*/
90 }
91 return 0;
92 }
93
94 /* ----- */
95 /* 主程序 */
96 /* ----- */
97 void main ()
98 {
99 int KeyValue; /* 欲查找数据变量 */
100 int Index; /* 输入数据索引 */
101 int i;
102
103 Index = 0;
104
105 printf("Input Data : "); /* 输出输入数据 */
106 for (i=0; i<Max; i++)
107 printf("[%d]", Data[i]);
108 printf("\n");
109
110 for (i=0; i<HashMax; i++) /* 杂凑表初始化 */
111 HashTab[i] = 0;
112
113 while (Index < Max)
114 {
115 if (Create_Hash(Data[Index]))
116 printf("Hash Success!!\n"); /* 杂凑建立成功 */
117 else
118 printf("Hash Filled!!\n"); /* 杂凑建立失败 */
119 Index++;
120 }
121
122 for (i=0; i<HashMax; i++) /* 输出杂凑表数据 */
123 {
124 printf("[%d]", HashTab[i]);
125 }
126 printf("\n");

```



```

127
128 while (KeyValue != -1) /* 输入-1 结束程序 */
129 {
130 printf(" Please enter your key value : ");
131 scanf("%d",&KeyValue);
132
133 if (Hash_Search(KeyValue))
134 printf("Search Time = %d\n",Counter); /* 输出查找次数
135 */
136 else
137 printf("No Found!!\n"); /* 输出没有找到数据 */
138 }
139 }

```

运行结果:

```

C:\DS>h_search
Input Data : [12][160][219][522][725][9997]
Key : 12 => Address 2
[0][0][12][0][0]
Hash Success!!
Key : 160 => Address 0
[160][0][12][0][0]
Hash Success!!
Key : 219 => Address 4
[160][0][12][0][219]
Hash Success!!
Collision 1 => Address 2
Collision 2 => Address 0
Key : 522 => Address 3
[160][0][12][522][219]
Hash Success!!
Collision 1 => Address 0
Collision 2 => Address 3
Key : 725 => Address 1
[160][725][12][522][219]
Hash Success!!
Collision 1 => Address 2
Collision 2 => Address 0
Collision 3 => Address 3
Collision 4 => Address 1
Collision 5 => Address 4
Hash Filled!!
[160][725][12][522][219]
Please enter your key value : 160
Search Time = 1
Please enter your key value : 725
Search Time = 3
Please enter your key value : 39
No Found!!
Please enter your key value : -1
No Found!!

C:\DS>

```

接下来, 我们再介绍链表的杂凑表应用。

程序实例:

运用余数法作为杂凑函数及链表解决法解决杂凑碰撞解决法设计杂凑表。

## 程序构思:

依余数法将数据存储至杂凑表中、发生杂凑碰撞时以差值解决法解决。

余数法:

Address = Key % Array\_Size

链表解决法:

在数据之后再连结出一个链表, 并将最后一个链表的节点设为 NULL。

数据查找时, 同时也先用余数法运算出数据地址, 若找不到运用链表解决法找出下一个数据地址, 直到数据全查找完为止。

## 程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: k_search.c */
03 /* 程序目的: 运用余数法作为杂凑函数及链表解决法解 */
04 /* 决杂凑碰撞解决法设计杂凑表。 */
05 /* Written By Kuo-Yu Huang. (WANT Studio.) */
06 /* ===== */
07 #include <stdlib.h>
08 #define Max 6
09 #define HashMax 5
10 int Data[Max] = { 1, 13, 20, 5, 12, 33}; /* 数据数组 */
11 struct List /* 声明列表结构 */
12 {
13 int Key; /* 键值 */
14 struct List *Next; /* 下一个节点 */
15 };
16 typedef struct List Node;
17 typedef Node *Link;
18
19 Node HashTab[HashMax]; /* 杂凑表 */
20 int Counter = 1; /* 计数器 */
21
22 /* ----- */
23 /* 杂凑函数之余数法 */
24 /* ----- */
25 int Hash_Mod(int Key)
26 {
27
28 return Key % HashMax; /* 返回 键值除以杂凑表大小取余数 */
29
30 }
31
32 /* ----- */
33 /* 建立杂凑表 */
34 /* ----- */
35 void Create_Hash(int Key)
36 {
37 Link Pointer; /* 列表指针 */
38 Link New; /* 新列表指针 */
39 int Index; /* 杂凑索引 */
40
41 New = (Link) malloc(sizeof(Node)); /* 内存配置 */
42 New->Key = Key; /* 键值 */
43 New->Next = NULL; /* 指向结束指针 */
44

```

```

45 Index = Hash_Mod(Key); /* 取得数据位置 */
46 Pointer = HashTab[Index].Next; /* 杂凑表起始指针 */
47 if (Pointer != NULL) /* 指针没指向结束指针时 */
48 {
49 New->Next = Pointer; /* 指向上一个指针 */
50 HashTab[Index].Next = New; /* 杂凑表指针指向此新指针 */
51 }
52 else
53 HashTab[Index].Next = New; /* 杂凑表指针指向此新指针 */
54 }
55
56 /* -----*/
57 /* 杂凑查找法 */
58 /* -----*/
59 int Hash_Search(int Key)
60 {
61 Link Pointer; /* 列表指针 */
62 int Index; /* 杂凑索引 */
63
64 Counter = 0;
65 Index = Hash_Mod(Key); /* 取得数据位置 */
66 Pointer = HashTab[Index].Next; /* 杂凑表起始指针 */
67 printf("Data[%d] : ", Index);
68
69 while (Pointer != NULL) /* 指针指向结束指针时终止循环 */
70 /*
71 {
72 Counter++;
73 printf("[%d]", Pointer->Key);
74 if (Pointer->Key == Key) /* 查找到数据 */
75 return 1;
76 else /* 往下一个指针查找 */
77 Pointer = Pointer->Next; /* 指向下一个指针 */
78 }
79 return 0;
80 }
81
82 /* -----*/
83 /* 主程序 */
84 /* -----*/
85 void main ()
86 {
87 int KeyValue; /* 欲查找数据变量 */
88 int Index; /* 输入数据索引 */
89 Link Pointer;
90 int i;
91
92 Index = 0;
93
94 printf("Input Data : "); /* 输出输入数据 */
95 for (i=0; i<Max; i++)
96 printf("[%d]", Data[i]);
97 printf("\n");
98
99 while (Index < Max) /* 建立杂凑表 */
100 {
101 Create_Hash(Data[Index]);
102 Index++;
103 }
104
105 for (i=0; i<HashMax; i++) /* 输出杂凑表数据 */

```

```

106 {
107 printf("HashTab[%d] : ",i);
108 Pointer = HashTab[i].Next;
109 while (Pointer != NULL)
110 {
111 if (Pointer->Key > 0)
112 printf("[%d]",Pointer->Key);
113 Pointer = Pointer->Next;
114 }
115 printf("\n");
116 }
117
118 while (KeyValue != -1) /* 输入-1 结束程序 */
119 {
120 printf("Please enter your key value : ");
121 scanf("%d",&KeyValue);
122
123 if (Hash_Search(KeyValue))
124 printf("\nSearch Time = %d\n",Counter); /* 输出查找次数 */
125 /*
126 else
127 printf("\nNo Found!!\n"); /* 输出没有找到数据 */
128 }
129 }

```

运行结果:

```

C:\DS>k_search
Input Data : [1][13][20][5][12][33]
HashTab[0] : [5][20]
HashTab[1] : [1]
HashTab[2] : [12]
HashTab[3] : [33][13]
HashTab[4] :
Please enter your key value : 20
Data[0] : [5][20]
Search Time = 2
Please enter your key value : 33
Data[3] : [33]
Search Time = 1
Please enter your key value : 50
Data[0] : [5][20]
No Found!!
Please enter your key value : -1
Data[-1] :
No Found!!

C:\DS>

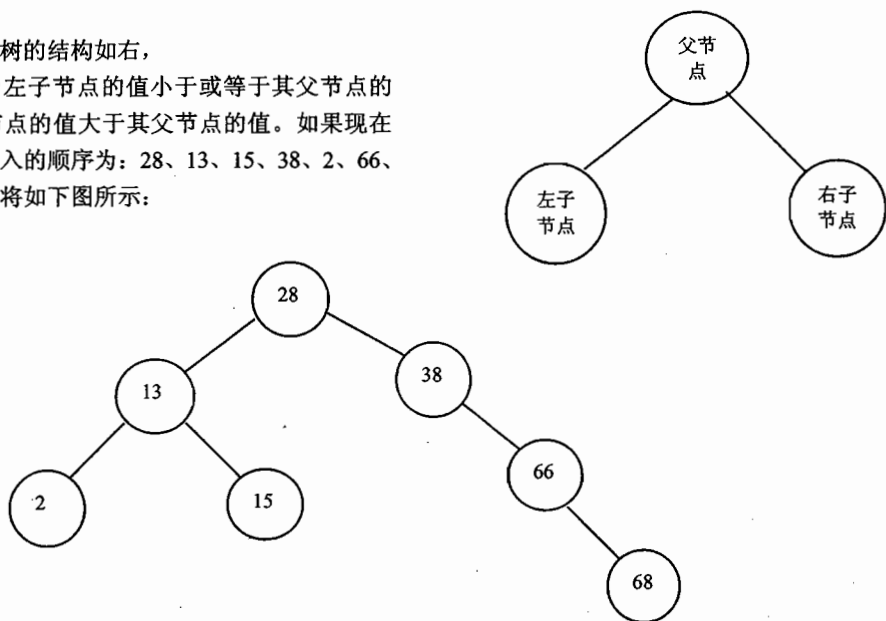
```

## 9.7 二叉查找树

我们之前的各小节皆以数组来存储数据，但就如同数组那章所介绍，数组结构数据的添加及删除会有许多问题。如果我们换一种数据结构来存储数据，再进行查找工作，那么数据的添加、删除上的问题就可以解决了。这一节我们的重点介绍用二叉查找树存储数据。

二叉查找树的结构如右，

其规则为左子节点的值小于或等于其父节点的值，而右子节点的值大于其父节点的值。如果现在有7笔数据输入的顺序为：28、13、15、38、2、66、68。其二叉树将如下图所示：



基于二叉查找树的规则，所以当我们欲查找某一个数值时，我们只要先比较其根部父节点，如果欲查找值比根部的父节点值小则往其左侧子树查找；如果欲查找值比根部的父节点值大则往其右侧子树查找。

如果我们现在想找出 15，则：

步骤 1：欲查找值 15 小于根部父节点 28，所以往其左侧子树查找。

此时根部父节点改为左侧子树的根部父节点 13。

步骤 2：欲查找值 15 大于根部父节点 13，所以往其右侧子树查找。

此时根部父节点改为右侧子树的根部父节点 15。

步骤 3：欲查找值 15 等于根部父节点 15，表示找到数据。

如果我们现在想找出 30，则：

步骤 1：欲查找值 30 大于根部父节点 28，所以往其右侧子树查找。

此时根部父节点改为右侧子树的根部父节点 38。

步骤 2：欲查找值 30 小于根部父节点 38，所以往其左侧子树查找。

此时左侧子树已达底部，而没有根部父节点。表示未能查找到数据。

二叉查找树，若不考虑其建立二叉查找树的时间，其时间复杂度为  $O(\log n)$ ，和折半查找法一样。

#### 程序实例：

设计二叉查找树的程序。

#### 程序构思：

先将所输入的数据建立成一根二叉查找树。

欲查找值 KeyValue 与根部父节点 Root 比较：

##### 1. KeyValue 小于 Root 时：

Root = 左侧子树的根部节点。

##### 2. KeyValue 大于 Root 时：

Root = 右侧子树的根部节点。

重复执行上述两个步骤直到找到欲查找数据或子树已达底部，而没有根部父节点为止。

#### 程序源代码：

```

01 /* ===== Program Description ===== */
02 /* 程序名称: t_search.c */
03 /* 程序目的: 设计二叉查找树的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define Max 10
08 int Data[Max] = { 15, 2, 13, 6, 17,
09 25, 37, 7, 3, 18}; /* 数据数组 */
10 int Counter; /* 计数器 */
11 struct Tree /* 声明树状结构 */
12 {
13 int Key; /* 数据变量 */
14 struct Tree *Left; /* 左子树指针 */
15 struct Tree *Right; /* 右子树指针 */
16 };
17 typedef struct Tree TreeNode;
18 typedef TreeNode *BTree;
19
20 BTree Root = NULL; /* 根部父节点 */
21
22 /* ----- */
23 /* 建立二叉查找树 */
24 /* ----- */
25 void Create_Tree(int *Data)
26 {
27 BTree New; /* 新指针变量 */
28 BTree Current; /* 目前指针 */
29 BTree Father; /* 父节点指针 */
30 int i;
31
32 for (i=0; i<Max; i++)
33 {
34 New = (BTree) malloc(sizeof(TreeNode)); /* 内存配置 */
35 New->Key = Data[i]; /* 新指针为输入的数据 */
36 New->Left = NULL; /* 左子树节点 */
37 New->Right = NULL; /* 右子树节点 */
38 if (Root == NULL) /* 当根节点未连结任何子树时 */

```

```

39 Root = New; /* 根节点为新节点 */
40 Else
41 {
42 Current = Root; /* 目前的位置在根节点 */
43 while (Current != NULL) /* 当目前的节点为最底端则结束循环 */
44 {
45 Father = Current; /* 记录父节点 */
46 if (Current->Key >= Data[i]) /* 目前节点数据大于或等于
47 输入数据 */
48 Current = Current->Left; /* 往左子树 */
49 else /* 目前节点数据小于输入数据 */
50 Current = Current->Right; /* 往右子树 */
51 }
52 if (Father->Key > Data[i]) /* 串起父与子节点 */
53 Father->Left = New;
54 Else
55 Father->Right = New;
56 }
57 }
58 }
59
60 /* -----*/
61 /* 二叉查找树 */
62 /* -----*/
63 int Tree_Search(int Key)
64 {
65 BTree Pointer;
66
67 Pointer = Root; /* 查找节点为根节点 */
68 Counter = 0;
69 while (Pointer != NULL)
70 {
71 Counter++;
72 /* 查找节点的数据小于欲查找数据 */
73 if (Pointer->Key < Key)
74 Pointer = Pointer->Right; /* 往右子树 */
75 /* 查找节点的数据大于欲查找数据 */
76 else if (Pointer->Key > Key)
77 Pointer = Pointer->Left; /* 往左子树 */
78 /* 查找节点的数据等于欲查找数据 */
79 else if (Pointer->Key == Key)
80 return 1; /* 找到数据 */
81 }
82 return 0; /* 未能查找到数据 */
83 }
84
85 /* -----*/
86 /* 主程序 */
87 /* -----*/
88 void main ()
89 {
90 int KeyValue; /* 欲查找数据变量 */
91 int i;
92
93
94 printf("Input Data : "); /* 输出输入数据 */
95 for (i=0; i<Max; i++)
96 printf("[%d]", Data[i]);
97 printf("\n");
98
99 Root = NULL;

```

```

100 Create_Tree(Data); /* 调用建立二叉查找树 */
101
102 while (KeyValue != -1) /* 输入-1 结束程序 */
103 {
104 printf("Please enter your key value : ");
105 scanf("%d",&KeyValue);
106
107 if (Tree_Search(KeyValue))
108 printf("Search Time = %d\n",Counter); /* 输出查找次数 */
109 else
110 printf("No Found!!\n"); /* 输出没有找到数据 */
111 }
112 }

```

运行结果:

```

C:\DS>t_search
Input Data : [15][2][13][6][17][25][37][7][3][18]
Please enter your key value : 17
Search Time = 2
Please enter your key value : 15
Search Time = 1
Please enter your key value : 37
Search Time = 4
Please enter your key value : 20
No Found!!
Please enter your key value : -1
No Found!!

C:\DS>

```

## 【习题】

### 一、复习:

1. 线性查找法不必考虑数据是否已经排序好,就可以进行数据查找工作。
2. 线性查找法的最坏状况时间复杂度为  $W(n)=n \in O(n)$ 。
3. 折半查找法的最坏状态的时间复杂度为  $W(n)=n \in O(n)$ 。
4. 有 30 笔已排序的数据若用折半查找法,最多只需要 4 次比较即可知道数据是否存在。
5. 插补查找法是以比例的概念,求出欲查找数据可能的位置,然后进行比较。
6. 插补查找法对于查找不均匀分布的数据较有效率。
7. 杂凑查找法每次查找只需一次即可找出数据。
8. 线性查找最大的优点是(复选):
  - (a) 数据愈多时,线性查找比折半查找法快。
  - (b) 数据愈少时,线性查找比折半查找法快。
  - (c) 数据无需事先排序,即可进行查找。
  - (d) 程序编写上很简单。
9. 使用二次方探索法来处理杂凑碰撞时,若第一次发生杂凑碰撞的位置在 5,数组的大小为 11,一连 3 次碰撞(即第 3 次碰撞时),新的数据位置为:
  - (a) 6
  - (b) 8



(c) 10

(d) 12

## 二、应用:

1. 有一个整数数组的数据内容如下:

|      |   |    |    |    |    |    |    |    |    |     |     |     |
|------|---|----|----|----|----|----|----|----|----|-----|-----|-----|
|      | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9   | 10  | 11  |
| Data | 3 | 15 | 35 | 36 | 39 | 52 | 57 | 66 | 89 | 158 | 160 | 350 |

试利用费氏查找法查找 39 和 180, 请写出运作过程。

2. 有一个整数数组的数据内容如下:

|      |   |   |    |    |    |    |    |    |    |    |    |    |
|------|---|---|----|----|----|----|----|----|----|----|----|----|
|      | 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| Data | 2 | 8 | 10 | 15 | 16 | 19 | 23 | 25 | 33 | 34 | 38 | 50 |

试利用插补查找法查找 10 和 36, 请写出运作过程。

3. 有一个整数数组的数据内容如下:

|      |   |    |    |    |    |    |    |     |     |     |     |     |
|------|---|----|----|----|----|----|----|-----|-----|-----|-----|-----|
|      | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7   | 8   | 9   | 10  | 11  |
| Data | 9 | 50 | 51 | 53 | 69 | 80 | 92 | 150 | 178 | 199 | 320 | 580 |

试利用加强型插补查找法查找 10 和 178, 请写出运作过程。

4. 试运用数值抽出法作为杂凑函数及二次方探索法解决杂凑碰撞解决法设计杂凑表。
5. 有一个整数数组的数据内容如下:

|      |    |    |    |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|----|----|----|
|      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| Data | 50 | 46 | 60 | 82 | 33 | 12 | 58 | 90 | 32 | 22 | 30 | 69 |

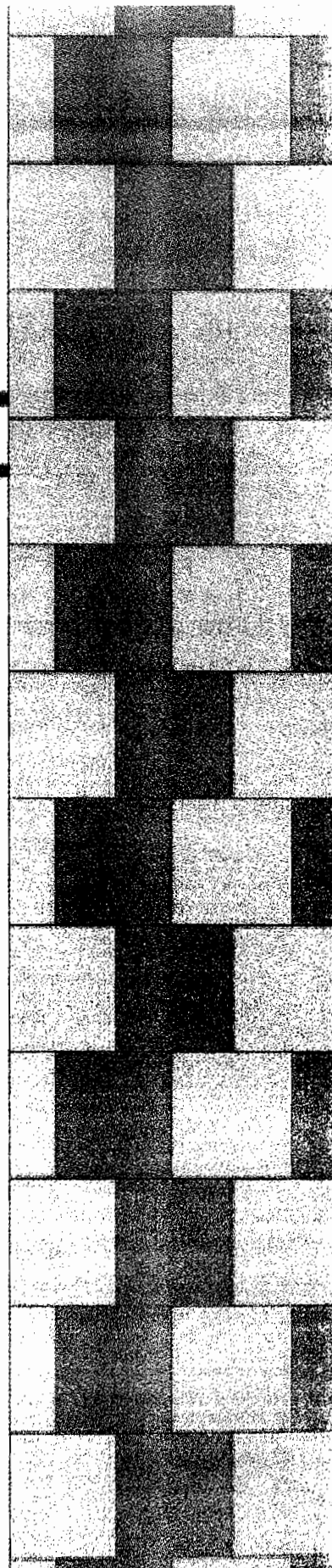
试建立其二叉查找树, 并列表算出各数据的查找次数。

# 高级链表

## 第 10 章

◆ 循环链表

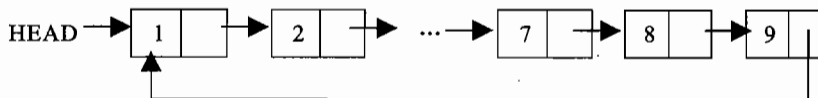
◆ 双链表



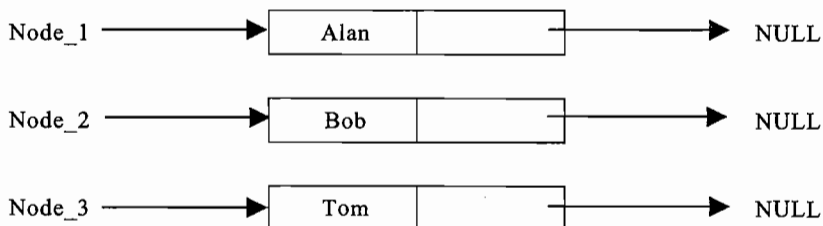
## 10.1 循环链表

### 10.1.1 循环链表的建立与释放

除了单向的链表外，我们还可以将链表做成环状的链表，也就是将链表的尾端指针指向首节点。在这一节，我们将把重点放在循环链表上，让读者对链表有更深入的了解。循环链表的结构如下图所示：

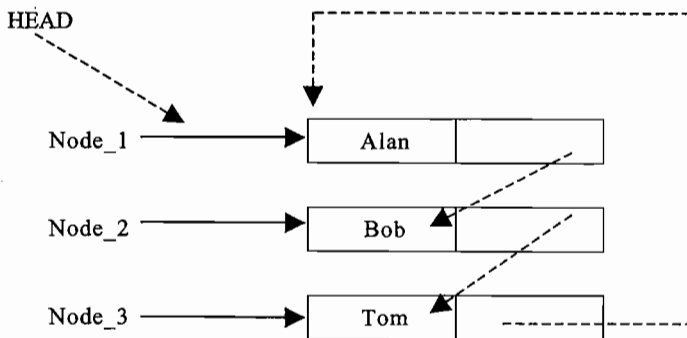


首先，我们将说明循环链表的建立。建立循环链表的方式和建立单链表类似，只不过单链表的最后一个节点指向 NULL，而循环链表的最后一个节点是指向首节点上。我们以建立单链表的例子介绍环状列表。现在有 3 个节点如下：

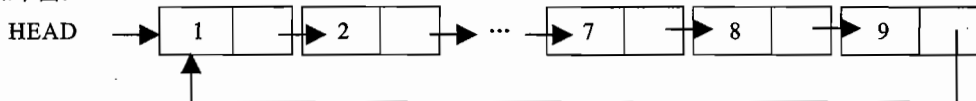


我们现在想要把 3 个节点依次给串连成循环链表，则运作过程如下：

```
HEAD = Node_1
Node_1->Next = Node_2
Node_2->Next = Node_3
Node_3->Next = HEAD
```



在循环链表的释放，如果采用和单链表相同的释放方式，会发生不知是否已到达最后一个节点的错误。如下图：



如果采用和单链表相同的内存释放方式，第一步将释放首节点(HEAD)的内存空间，一直释放最后一个节点，可是最后一个节点的指针却指向首节点，但首节点却早已从内存中释放，如此一来，造成最后一个节点的指针所指的位置是一个内存早已不存在的数据。所以我们必须改变一下做法，从思考的过程中，我们可采用下列两种方式：

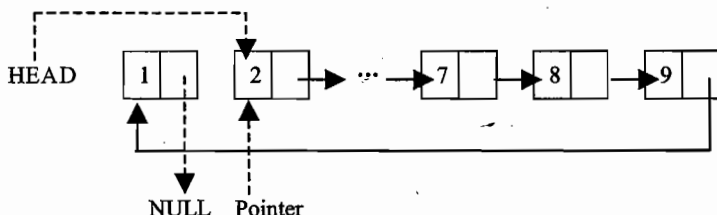
第1种方式：将循环链表改为单链表，即可用单链表释放的方式。

第1步将 Pointer 节点指向首节点的指针所指的节点(即第2个节点)。

第2步将首节点的指针指向 NULL。

如此一来便成为了一个单向的链表。一个以 Pointer 节点为首节点的单链表。

```
Pointer = Head->Next;
Head->Next = NULL;
Head = Pointer;
Free_List(Head);
```



第2种方式：从循环链表的第2个节点开始释放。

第1步将 Pointer 节点指向首节点的指针所指的节点(即第2个节点)。第2步持续往下一个节点释放，直到节点是首节点为止。最后再将首节点释放。

```
Next = Head->Next;
while (Next != Head)
{
 Pointer = Next;
 Next = Next->Next;
 free(Pointer);
}
free_(Head);
```

从这个例子中，不知读者是否学习了些什么呢？在程序设计的领域中，每一个问题的答案常常不是只有一个唯一的解而已，用那一种方式来设计比较好，常常没有绝对的答案，一种解法的好坏也是见仁见智的。希望读者从思考的过程中，找出解答，多看看别人对同一个问题不同的解法，才是增进程序设计水平的不二法门。

程序实例：

设计一个将输入的数据建立成循环链表、输出循环链表数据，并释放循环链表。

## 程序构思:

## 循环链表的建立:

先声明一个首节点 Head, 并将 Head->Next 设为 NULL。

每输一笔数据就声明一个新节点 New, 把 New->Next 设为 NULL, 并且链接到之前列表的尾端。

最后一个节点的指针则指向首节点 HEAD。

## 循环链表数据的输出:

先将 Pointer 节点的指针指向第一个节点, 将 Pointer 节点(即第 1 个节点)的数据输出。

然后再将 Pointer 节点的指针指向 Pointer 指针的指针(即下 1 节点), 将 Pointer 节点(即第 1 个节点)的数据输出。重复执行此步骤直到 Pointer 节点等于首节点 HEAD 为止。

## 链接串状链表的释放:

将 Pointer 节点指向首节点的指针所指的节点(即第 2 个节点)。持续往下一个节点释放, 直到节点是首节点为止。最后再将首节点释放。

## 程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: c_create.c */
03 /* 程序目的: 设计一个将输入的数据建立成循环链表、 */
04 /* 输出循环链表数据, 并释放循环链表 */
05 /* Written By Kuo-Yu Huang. (WANT Studio.) */
06 /* ===== */
07 #include <stdlib.h>
08 #define Maxl 10
09 struct List /* 节点结构声明 */
10 {
11 int Number;
12 struct List *Next;
13 };
14 typedef struct List Node;
15 typedef Node *Link;
16 /* ----- */
17 /* 释放循环链表 */
18 /* ----- */
19 void Free_CList(Link Head)
20 {
21 Link Pointer; /* 节点声明 */
22 Link Next; /* 节点声明 */
23
24 Next = Head->Next; /* 下一个节点为首节点的下一个节点 */
25 while (Next != Head) /* 当下一个节点为首节点时, 结束循环 */
26 {
27 Pointer = Next;
28 Next = Next->Next; /* 往下一个节点 */
29 free(Pointer);
30 }
31 free(Head);
32 }
33
34 int Datal[Maxl] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
35 /* ----- */
36 /* 输出循环链表数据 */
37 /* ----- */
38 void Print_CList(Link Head)

```

```
39 {
40 Link Pointer; /* 节点声明 */
41 Pointer = Head; /* Pointer 指针设为首节点 */
42 printf("Input Data :\n");
43
44 do
45 {
46 printf("[%d]", Pointer->Number);
47 Pointer = Pointer->Next; /* 指向下一个节点 */
48 } while (Pointer != Head); /* 当节点为开头节点时，结束循环 */
49 /*
50 printf("\n");
51 */
52
53 /* ----- */
54 /* 建立循环链表 */
55 /* ----- */
56 Link Create_CList(Link Head,int *Data,int Max)
57 {
58 Link New; /* 节点声明 */
59 Link Pointer; /* 节点声明 */
60 int i;
61
62 Head = (Link) malloc(sizeof(Node)); /* 内存配置 */
63
64 if (Head == NULL) /* 内存配置失败 */
65 printf("Memory allocate Failure!!\n");
66 else
67 {
68 Head->Number = Data[0]; /* 定义首节点数据编号 */
69 Head->Next = NULL;
70
71 Pointer = Head; /* Pointer 指针设为首节点 */
72
73 for (i=1;i<Max;i++)
74 {
75 /* 内存配置 */
76 New = (Link) malloc(sizeof(Node));
77
78 New->Number = Data[i];
79 New->Next = NULL;
80 /* 将新节点串连在原列表尾端 */
81 Pointer->Next = New;
82 /* 列表尾端节点为新节点 */
83 Pointer = New;
84 }
85 Pointer->Next = Head; /* 将最后一个节点指向首节点 */
86 }
87 return Head;
88 }
89 /* ----- */
90 /* 主程序 */
91 /* ----- */
92 void main ()
93 {
94 Link Head; /* 节点声明 */
95
96 Head = Create_CList(Head,Datal,Max1); /* 调用建立循环链表 */
97
98 if (Head != NULL)
99 Print_CList(Head); /* 调用输出循环链表数据 */
```



```
100 Free CList(Head);
101 }
```

运行结果:

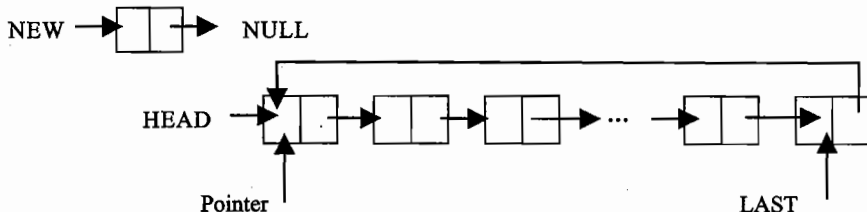
```
C:\DS>c_create
Input Data :
[1][2][3][4][5][6][7][8][9][0]

C:\DS>
```

## 10.1.2 循环链表内节点的插入

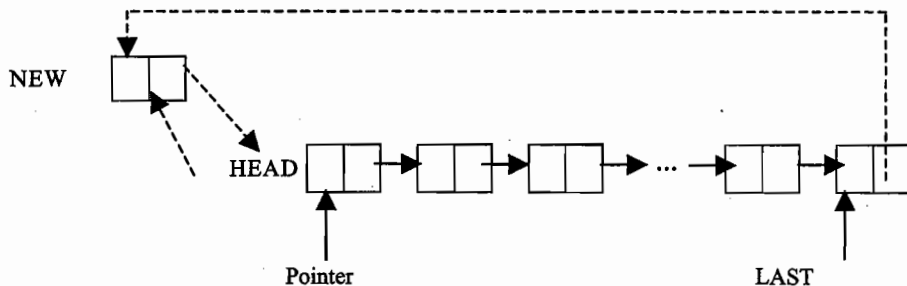
循环链表内的节点插入，我们可以分为几个情形来讨论：

### 1. 插入在循环链表开头之前

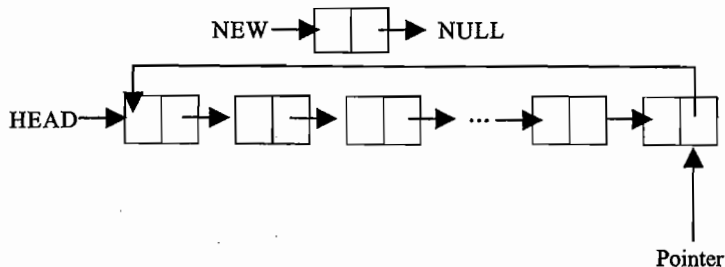


如果新的节点插入在循环链表的开始。第一步需要将新节点的指针指向循环链表的首节点，第二步将循环链表的最后一个节点的指针指向新的节点，最后再将首节点设为新节点。

```
NEW->Next = HEAD
LAST->Next = NEW
HEAD = NEW
```



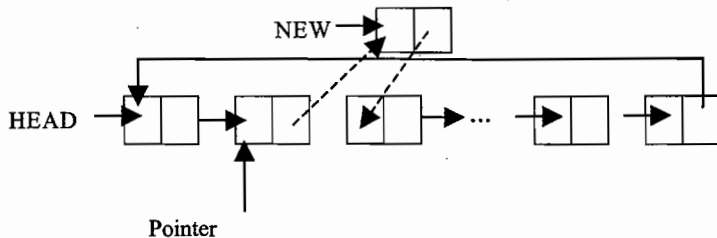
### 2. 插入在循环链表中间



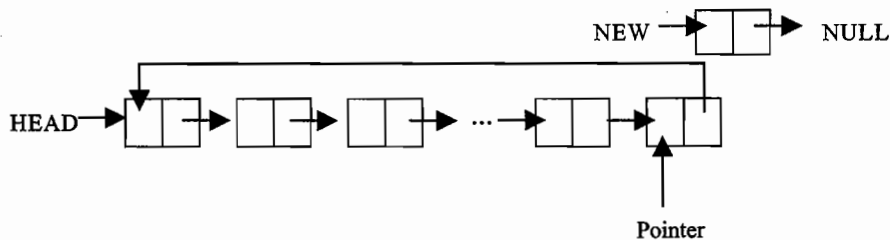
如果新的节点插入在循环链表的中间，我们可找到 'Pointer' 节点，则需要将新节点的指针指向 'Pointer'

节点的指针(即下一个节点),但不能让链表断裂。所以第一步,必须将新节点的指针指向 Pointer 节点的指针、第二步再将 Pointer 节点的指针指向新节点。

```
NEW->Next = Pointer->Next
Pointer->Next = NEW
```

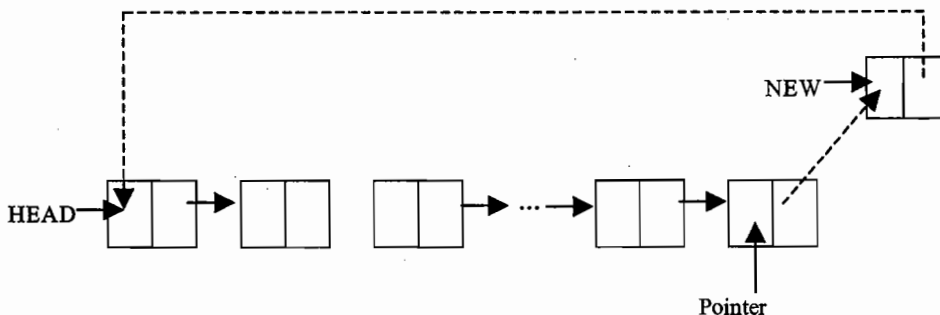


### 3. 插入在循环链表尾端



新的节点插入在循环链表的尾端(Pointer 节点),所以第一步,必须将新节点的指针指向 Pointer 节点的指针(即首节点 HEAD)、第二步再将 Pointer 节点的指针指向新节点。

```
NEW->Next = Pointer->Next
Pointer->Next = NEW
```



### 程序实例:

设计一个循环链表内节点插入的程序。

### 程序构思:

声明一个新节点供用户输入欲插入节点的内容。

用户输入一个节点内容,表示欲插入在那一个节点之后。

持续往下一个节点,直到节点内容等于 Key 或节点指针为 Head(即找不到该节点)。

如果找到该节点,且该节点的指针是首节点,则插入在环状列表尾端之后:

```
NEW->Next = Pointer->Next
```



```
Pointer->Next = NEW
```

如果找到该节点, 且该节点的指针不是首节点(Head), 则插入在环状列表中间:

```
NEW->Next = Pointer->Next
Pointer->Next = NEW
```

如果找不到, 则插入在首节点之前, 即首节点之前:

```
NEW->Next = HEAD
LAST->Next = NEW
HEAD = NEW
```

程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: c_create.c */
03 /* 程序目的: 设计一个将输入的数据建立成循环链表、 */
04 /* 并输出循环链表数据。 */
05 /* Written By Kuo-Yu Huang. (WANT Studio.) */
06 /* ===== */
07 #include <stdlib.h>
08 #define Max1 10
09 struct List /* 节点结构声明 */
10 {
11 int Number;
12 struct List *Next;
13 };
14 typedef struct List Node;
15 typedef Node *Link;
16
17 int Data1[Max1] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
18 /* ----- */
19 /* 释放循环链表 */
20 /* ----- */
21 void Free_CList(Link Head)
22 {
23 Link Pointer; /* 节点声明 */
24 Link Next; /* 节点声明 */
25
26 Next = Head->Next; /* 下一个节点为首节点的下一个节点 */
27 while (Next != Head) /* 当下一个节点为首节点时, 结束循环 */
28 {
29 Pointer = Next;
30 Next = Next->Next; /* 往下一个节点 */
31 free(Pointer);
32 }
33 free(Head);
34 }
35
36 /* ----- */
37 /* 插入节点至链表内 */
38 /* ----- */
39 Link Insert_CList(Link Head, Link New, int Key)
40 {
41 Link Pointer; /* 节点声明 */
42 Link Last;
43
44 Pointer = Head; /* Pointer 指针设为首节点 */
45
46 while (1)
47 {
```

```

48 /* 插入在链表中间或尾端 */
49 if (Pointer->Number == Key)
50 {
51 New->Next = Pointer->Next;
52 Pointer->Next = New;
53 break;
54 }
55 /* 找不到数据, 插入在链表首节点前 */
56 if (Pointer->Number != Key && Pointer->Next == Head)
57 {
58 New->Next = Pointer->Next;
59 Pointer->Next = New;
60 Head = New;
61 break;
62 }
63 Pointer = Pointer->Next; /* 往下一个节点 */
64 }
65 return Head;
66 }
67
68 /* -----*/
69 /* 输出循环链表数据 */
70 /* -----*/
71 void Print_CList(Link Head)
72 {
73 Link Pointer; /* 节点声明 */
74 Pointer = Head; /* Pointer 指针设为首节点 */
75 printf("Input Data :\n");
76
77 do
78 {
79 printf("[%d]", Pointer->Number);
80 Pointer = Pointer->Next; /* 指向下一个节点 */
81 } while (Pointer != Head); /* 当节点为开头节点时, 结束循环 */
82
83 printf("\n");
84 }
85
86 /* -----*/
87 /* 建立循环链表 */
88 /* -----*/
89 Link Create_CList(Link Head, int *Data, int Max)
90 {
91 Link New; /* 节点声明 */
92 Link Pointer; /* 节点声明 */
93 int i;
94
95 Head = (Link) malloc(sizeof(Node)); /* 内存配置 */
96
97 if (Head == NULL) /* 内存配置失败 */
98 printf("Memory allocate Failure!!\n");
99 else
100 {
101 Head->Number = Data[0]; /* 定义首节点数据编号 */
102 Head->Next = NULL;
103
104 Pointer = Head; /* Pointer 指针设为首节点 */
105
106 for (i=1; i<Max; i++)
107 {
108 /* 内存配置 */

```

```

109 New = (Link) malloc(sizeof(Node));
110
111 New->Number = Data[i];
112 New->Next = NULL;
113 /* 将新节点串连在原列表尾端 */
114 Pointer->Next = New;
115 /* 列表尾端节点为新节点 */
116 Pointer = New;
117 }
118 Pointer->Next = Head; /* 将最后一个节点指向首节点 */
119 }
120 return Head;
121 }
122
123 /* ----- */
124 /* 主程序 */
125 /* ----- */
126 void main ()
127 {
128 Link Head; /* 节点声明 */
129 Link New;
130 int Key;
131
132 Head = Create_CList(Head, Data1, Max1); /*调用建立环状链接串行*/
133
134 if (Head != NULL)
135 {
136 Print_CList(Head); /* 调用输出循环链表数据 */
137 while (1)
138 {
139 printf("Input 0 to EXIT\n"); /* 数据输入提示 */
140 New = (Link) malloc(sizeof(Node)); /* 内存配置 */
141 printf("Please input the data : ");
142 scanf("%d", &New->Number);
143 if (New->Number == 0) /* 输入 0 时结束循环 */
144 break;
145 printf("Please input the data number for Insert : ");
146 scanf("%d", &Key);
147
148 Head = Insert_CList(Head, New, Key); /* 调用插入节点
149 */
150 Print_CList(Head); /* 输出链表数据 */
151 }
152 }
153
154 Free_CList(Head);
155 }
156

```

运行结果:

```

C:\DS>c_insert
Input Data :
[1][2][3][4][5][6][7][8][9][0]
Input 0 to EXIT
Please input the data : 80
Please input the data number for Insert : 10
Input Data :
[80][1][2][3][4][5][6][7][8][9][0]
Input 0 to EXIT

```

```

Please input the data : 81
Please input the data number for Insert : 3
Input Data :
[80][1][2][3][81][4][5][6][7][8][9][0]
Input 0 to EXIT
Please input the data : 82
Please input the data number for Insert : 0
Input Data :
[80][1][2][3][81][4][5][6][7][8][9][0][82]
Input 0 to EXIT
Please input the data : 0

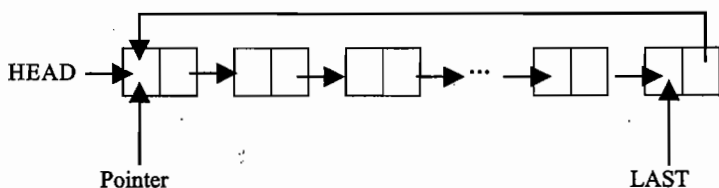
C:\DS>

```

### 10.1.3 循环链表内节点的删除

循环链表内节点的删除，依欲删除节点的位置来分，可分为下列几类：

#### 1. 删除循环链表首节点

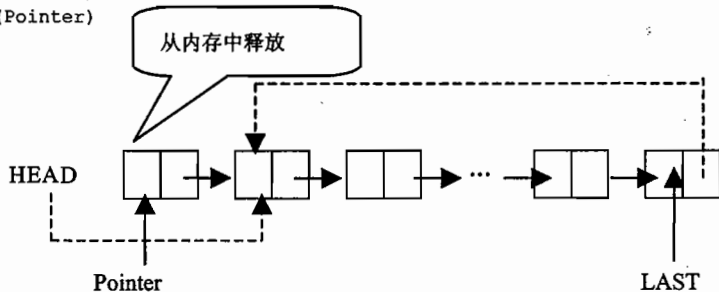


如果欲删除的节点是循环链表的首节点。第一步需要将首节点指向首节点的指针(即下一个节点)，第二步将循环链表的最后一个节点的指针指向首节点，最后再将原来的节点从内存中释放。

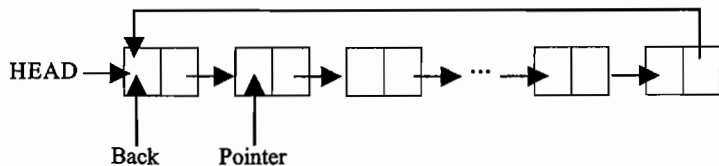
```

HEAD = Pointer->Next
LAST->Next = HEAD
Free(Pointer)

```



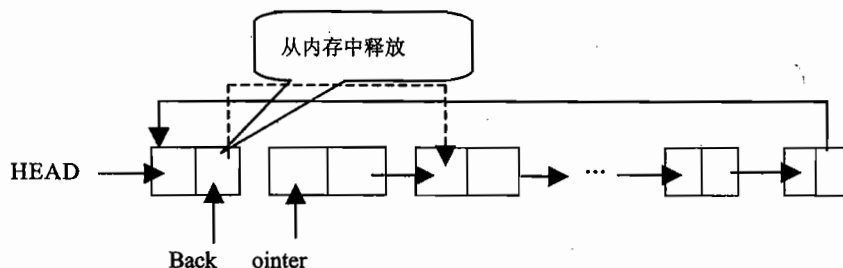
#### 2. 删除循环链表中间节点



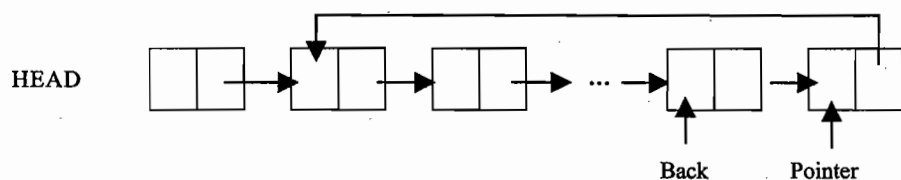
欲删除的节点在循环链表的中间，如果我们找到 Pointer 节点，则需要将前一个节点的指针指向 Pointer

节点的指针(即下一个节点), 并将原来的节点从内存中释放。

```
Back->Next = Pointer->Next
Free(Pointer)
```

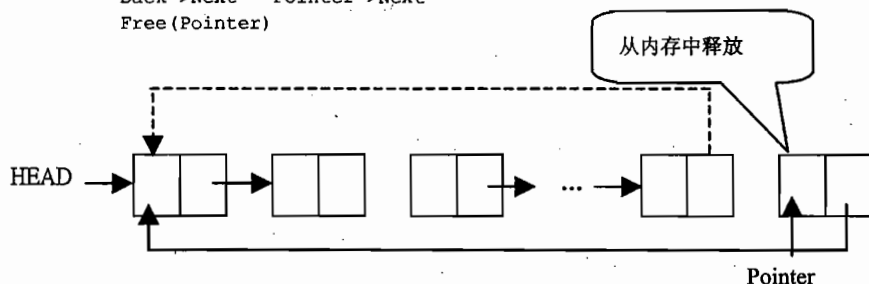


### 3. 删除循环链表尾端的节点



删除的节点在链表的尾端(Pointer 节点), 如果我们找到 Pointer 节点, 则需要将前一个节点的指针指向 Pointer 节点的指针(即为首节点)。并将原来的节点从内存中释放。

```
Back->Next = Pointer->Next
Free(Pointer)
```



### 程序实例:

设计一个删除循环链表内节点的程序。

### 程序构思:

持续往下一个节点查找欲删除节点, 直到节点内容找到或节点指针为 HEAD(即找不到该节点)。在删除时, 必须记录前一个节点的位置。

如果该节点不存在, 输出消息说节点不存在。

如果该节点存在, 且是首节点, 则必须找到尾端的节点:

```
HEAD = Pointer->Next
LAST->Next = HEAD
Free(Pointer)
```

如果该节点存在, 但非首节点(即链列表中节点或尾端节点), 则:

```
Back->Next = Pointer->Next
Free(Pointer)
```

程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: c_delete.c */
03 /* 程序目的: 设计一个删除循环链表内节点的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define Max1 10
08 struct List /* 节点结构声明 */
09 {
10 int Number;
11 struct List *Next;
12 };
13 typedef struct List Node;
14 typedef Node *Link;
15
16 int Data1[Max1] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
17 /* ----- */
18 /* 释放循环链表 */
19 /* ----- */
20 void Free_CList(Link Head)
21 {
22 Link Pointer; /* 节点声明 */
23 Link Next; /* 节点声明 */
24
25 Next = Head->Next; /* 下一个节点为首节点的下一个节点 */
26 while (Next != Head) /* 当下一个节点为首节点时, 结束循环 */
27 {
28 Pointer = Next;
29 Next = Next->Next; /* 往下一个节点 */
30 free(Pointer);
31 }
32 free(Head);
33 }
34
35 /* ----- */
36 /* 删除循环链表内节点 */
37 /* ----- */
38 Link Delete_CList(Link Head, int Key)
39 {
40 Link Pointer; /* 节点声明 */
41 Link Back; /* 上一个节点声明 */
42 Link Last; /* 尾端节点声明 */
43
44 Pointer = Head; /* Pointer 指针设为首节点 */
45
46 while (1)
47 {
48 /* 找不到数据 */
49 if ((Pointer->Number != Key) && (Pointer->Next == Head))
50 {
51 printf("Not Found!!\n");
52 break;
53 }
54 if (Head->Number == Key) /* 删除首节点 */
55 {
```

```

56 Last = Head;
57 while (Last->Next != Head)
58 Last = Last->Next;
59 Head = Pointer->Next;
60 Last->Next = Head;
61 free(Pointer);
62 break;
63 }
64 Back = Pointer;
65 Pointer = Pointer->Next; /* 往下一个节点 */
66 if (Pointer->Number == Key) /* 删除链表中中间或尾端节点 */
67 {
68 Back->Next = Pointer->Next;
69 free(Pointer);
70 break;
71 }
72 }
73 return Head;
74 }
75
76 /* ----- */
77 /* 输出循环链表数据 */
78 /* ----- */
79 void Print_CList(Link Head)
80 {
81 Link Pointer; /* 节点声明 */
82 Pointer = Head; /* Pointer 指针设为首节点 */
83
84 printf("Input Data :\n");
85 do
86 {
87 printf("[%d]",Pointer->Number);
88 Pointer = Pointer->Next; /* 指向下一个节点 */
89 } while (Pointer != Head); /* 当节点为开头节点时, 结束循环 */
90 printf("\n");
91 }
92
93 /* ----- */
94 /* 建立循环链表 */
95 /* ----- */
96 Link Create_CList(Link Head,int *Data,int Max)
97 {
98 Link New; /* 节点声明 */
99 Link Pointer; /* 节点声明 */
100 int i;
101
102 Head = (Link) malloc(sizeof(Node)); /* 内存配置 */
103
104 if (Head == NULL) /* 内存配置失败 */
105 printf("Memory allocate Failure!!\n");
106 else
107 {
108 Head->Number = Data[0]; /* 定义首节点数据编号 */
109 Head->Next = NULL;
110
111 Pointer = Head; /* Pointer 指针设为首节点 */
112
113 for (i=1;i<Max;i++)
114 {
115 /* 内存配置 */
116 New = (Link) malloc(sizeof(Node));

```

```

117
118 New->Number = Data[i];
119 New->Next = NULL;
120 /* 将新节点串连在原列表尾端 */
121 Pointer->Next = New;
122 /* 列表尾端节点为新节点 */
123 Pointer = New;
124 }
125 Pointer->Next = Head; /* 将最后一个节点指向首节点 */
126 }
127 return Head;
128 }
129
130 /* ----- */
131 /* 主程序 */
132 /* ----- */
133 void main ()
134 {
135 Link Head; /* 节点声明 */
136 int Key;
137
138 Head = Create_CList(Head,Data1,Max1); /* 调用建立循环链表 */
139
140 if (Head != NULL)
141 {
142 Print_CList(Head); /* 调用输出循环链表数据 */
143 while (1)
144 {
145 printf("Input 0 to EXIT\n"); /* 数据输入提示 */
146 printf("Please input the data number for Delete : ");
147 scanf("%d",&Key);
148 if (Key == 0) /* 输入 0 时结束循环 */
149 break;
150
151 Head = Delete_CList(Head,Key); /* 调用插入节点 */
152 Print_CList(Head); /* 调用输出循环链表数据 */
153 }
154 }
155 Free_CList(Head); /* 调用释放循环链表 */
156 }

```

运行结果:

```

C:\DS>c_delete
Input Data :
[1][2][3][4][5][6][7][8][9][10]
Input 0 to EXIT
Please input the data number for Delete : 1
Input Data :
[2][3][4][5][6][7][8][9][10]
Input 0 to EXIT
Please input the data number for Delete : 5
Input Data :
[2][3][4][6][7][8][9][10]
Input 0 to EXIT
Please input the data number for Delete : 10
Input Data :
[2][3][4][6][7][8][9]
Input 0 to EXIT
Please input the data number for Delete : 0

```



C:\DS&gt;

## 10.2 双链表

### 10.2.1 双链表的建立与释放

之前所介绍的链表都只是单链表，我们仅能从现在的节点指针前往下一个节点，而无法从现在的节点返回前一个节点。这一节我们所要介绍的是一种可能返回前一个节点的链表，我们称为双链表，其节点的结构如下图所示：

| Data | Back | Next |
|------|------|------|
|      |      |      |

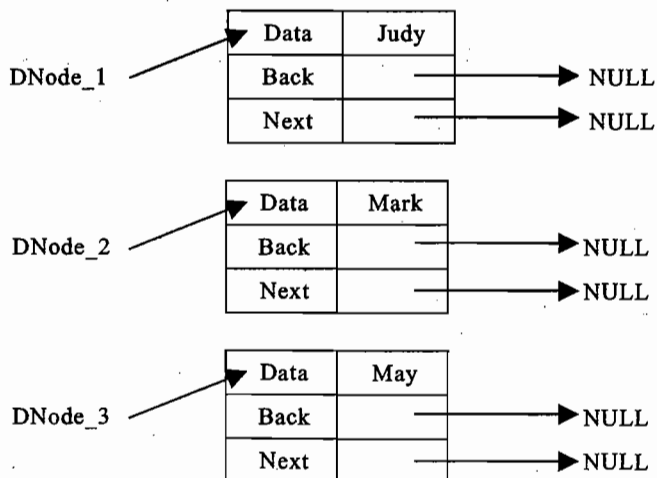
每一个双链表的节点包含了数据部分、指往链表上一个节点的指针和指往链表下一个节点的指针。在 C 语言中，双链表节点结构的声明格式如下：

```

struct DList
{
 数据类型 数据变量;
 struct DList *Back;
 struct Dlist *Next;
};
typedef struct DList Node;
typedef Node *DLink;

```

例如：现在有 3 个双链表的节点如下：



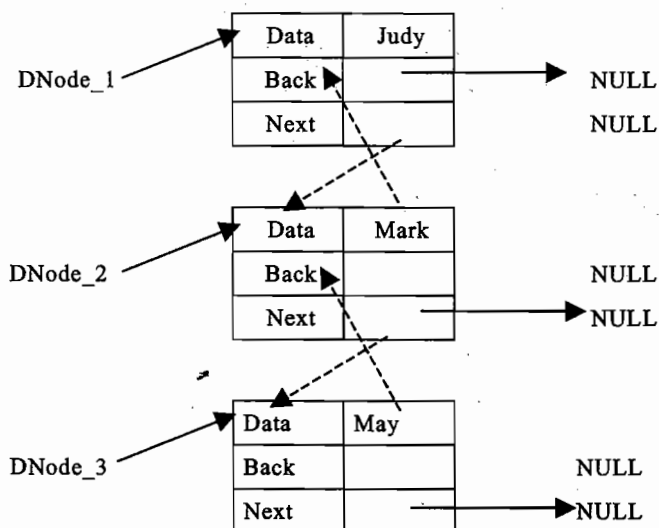
我们想将这 3 个双链表的节点，串连成双链表，其运作过程如下：

```

DNode_1->Next = DNode_2
DNode_2->Back = DNode_1
DNode_2->Next = DNode_3

```

DNode-3->Back = DNode\_2



而双链表的释放和单链表的释放的方式一样。从首节点开始一个一个的将节点释放，直到下一个节点指针指向 NULL(即到达双链表的尾端)为止。

#### 程序实例：

设计一个将输入的数据建立成双链表、输出双链表数据，并在程序结束后将双链表释放。

#### 程序构思：

双链表的建立：

先声明一个双链表的首节点 Head，并将 Head->Next 和 Head->Back 设为 NULL。

每输一笔数据就声明一个新节点 New，把 New->Next 和 New->Back 设为 NULL，并且链接到之前列表的尾端，再将 New->Back 指向原列表的尾端。

双链表数据的输出(和单链表相同)：

先将 Pointer 节点的指针指向第一个节点，将 Pointer 节点(即第一个节点)的数据输出。

然后再将 Pointer 节点的 Next 指针指向 Pointer 节点 Next 指针的 Next 指针(即下一节点)，将 Pointer 节点的数据输出。重复执行此步骤直到 Pointer 指针指向 NULL 为止。

双链表的释放(和单链表相同)：

先将 Pointer 节点的指针指向第一个节点，然后再将首节点(Head)设为首节点的指针(即下一节点)，将 Pointer 节点(即第一个节点)释放。重复执行此步骤直到首节点的指针指向 NULL 为止。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: d_create.c */
03 /* 程序目的: 设计一个将输入的数据建立成双链表、 */
04 /* 输出双链表数据, 并在程序结束后将双 */
05 /* 向链表释放。 */
06 /* Written By Kuo-Yu Huang. (WANT Studio.) */
07 /* ===== */
08 #include <stdlib.h>
09 #define Maxl 10
10 struct Dlist /* 节点结构声明 */
11 {
12 int Number;
13 struct Dlist *Back;
14 struct Dlist *Next;
15 };
16 typedef struct Dlist DNode;
17 typedef DNode *DLink;
18
19 int Data1[Maxl] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
20
21 /* ----- */
22 /* 释放双链表 */
23 /* ----- */
24 void Free_DList(DLink Head)
25 {
26 DLink Pointer; /* 节点声明 */
27
28 while (Head != NULL) /* 当节点为 NULL 结束循环 */
29 {
30 Pointer = Head;
31 Head = Head->Next; /* 往下一个节点 */
32 free(Pointer);
33 }
34 }
35
36 /* ----- */
37 /* 输出双链表数据 */
38 /* ----- */
39 void Print_DList(DLink Head)
40 {
41 DLink Pointer; /* 节点声明 */
42 Pointer = Head; /* Pointer 指针设为首节点 */
43
44 printf("Input Data : \n");
45 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
46 {
47 printf("%d", Pointer->Number);
48 Pointer = Pointer->Next; /* 往下一个节点 */
49 }
50 }
51
52 /* ----- */
53 /* 建立双链表 */
54 /* ----- */
55 DLink Create_DList(DLink Head, int *Data, int Max)
56 {
57 DLink New; /* 节点声明 */
58 DLink Pointer; /* 节点声明 */

```

```

59 int i;
60
61 Head = (DLink) malloc(sizeof(DNode)); /* 内存配置 */
62
63 if (Head == NULL) /* 内存配置失败 */
64 printf("Memory allocate Failure!!\n");
65 else
66 {
67 Head->Number = Data[0]; /* 定义首节点数据编号 */
68 Head->Back = NULL;
69 Head->Next = NULL;
70
71 Pointer = Head; /* Pointer 指针设为首节点 */
72
73 for (i=1;i<Max;i++)
74 {
75 /* 内存配置 */
76 New = (DLink) malloc(sizeof(DNode));
77
78 New->Number = Data[i];
79 New->Back = NULL;
80 New->Next = NULL;
81 /* 将新节点串连在原列表尾端 */
82 Pointer->Next = New;
83 /* 新节点的前一个节点为原列表尾端 */
84 New->Back = Pointer;
85 /* 列表尾端节点为新节点 */
86 Pointer = New;
87 }
88 }
89 return Head;
90 }
91
92 /* ----- */
93 /* 主程序 */
94 /* ----- */
95 void main ()
96 {
97 DLink Head; /* 节点声明 */
98
99 Head = Create_DList(Head,Data1,Max1); /* 调用建立循环链表 */
100
101 if (Head != NULL)
102 Print_DList(Head); /* 调用输出循环链表数据 */
103 Free_DList(Head);
104 }

```

运行结果:

```

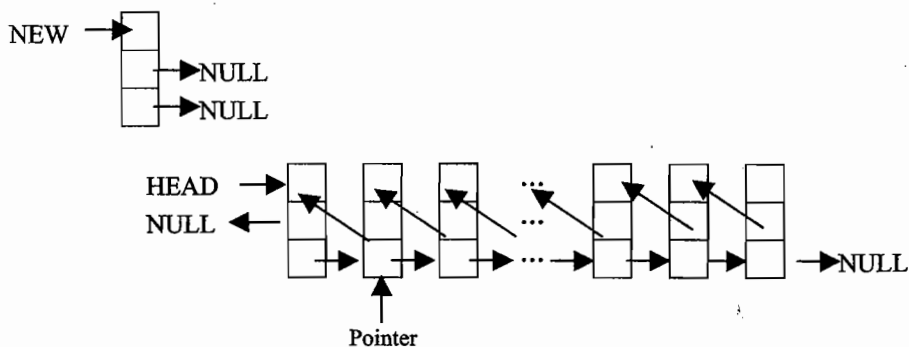
C:\tc>d_create
Input Data :
[1][2][3][4][5][6][7][8][9][0]
C:\tc>

```

## 10.2.2 双链表的插入

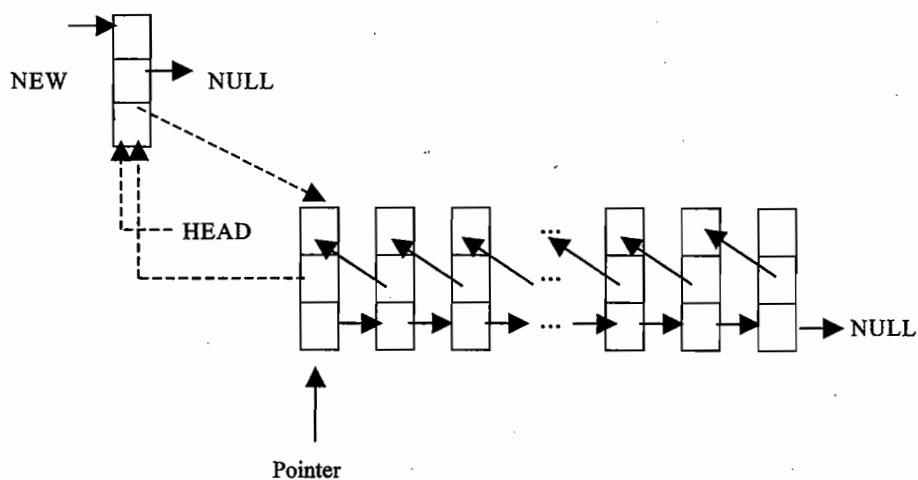
双链表的插入,大致上与单链表的插入相同,只不过双向链接的 Back 指针需要在新节点插入后,做些调整。双链表的节点的插入,可分为下列几类:

## 1. 插入在双链表首节点前

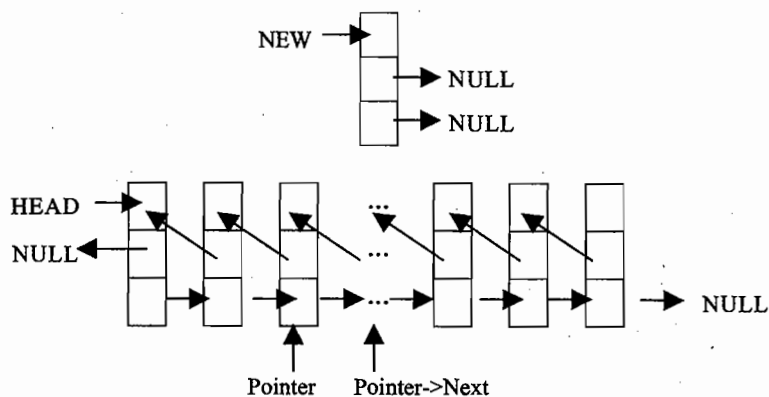


新的节点插入在双链表的开始, 需要将新节点的 Next 指针指向双链表的首节点, 并将双链表的首节点的 Back 指针指向新节点, 最后再将双链表的首节点设为新节点。

```
NEW->Next = Pointer
Pointer->Back = New
HEAD = New
```

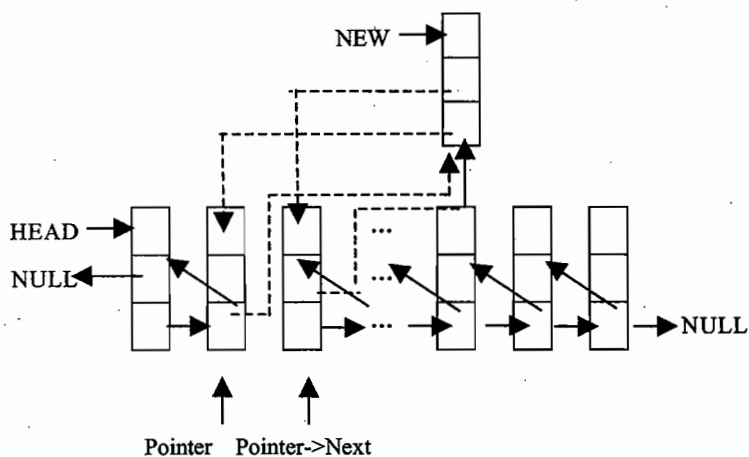


## 2. 插入在双链表中间

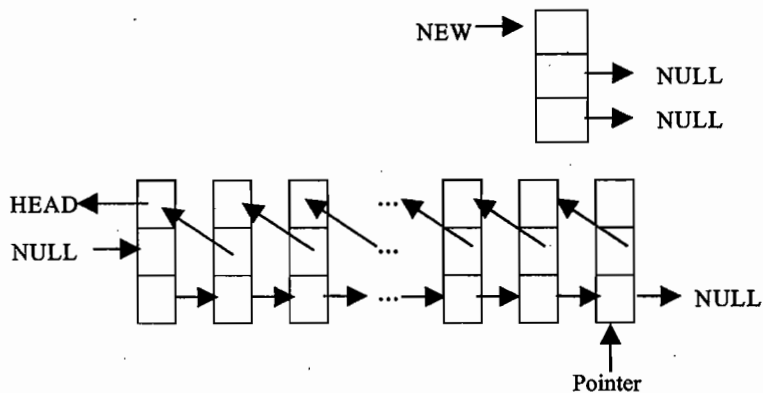


新的节点插入在双链表的中间, 如果我们找到 Pointer 节点, 则需要将新节点的指针指向 Pointer 节点的指针(即下一个节点), 但不能让链表断裂。所以第一步, 必须将新节点的 Next 指针指向 Pointer 节点的指针、第二步再将新节点的 Back 指针指向 Pointer 节点, 第 3 步将 Pointer 的 Next 指针指向新节点, 最后再将新节点 Next 指针的 Back 指针指向新节点。

```
NEW->Next = Pointer->Next
NEW->Back = Pointer
Pointer->Next = NEW
New->Next->Back = New
```

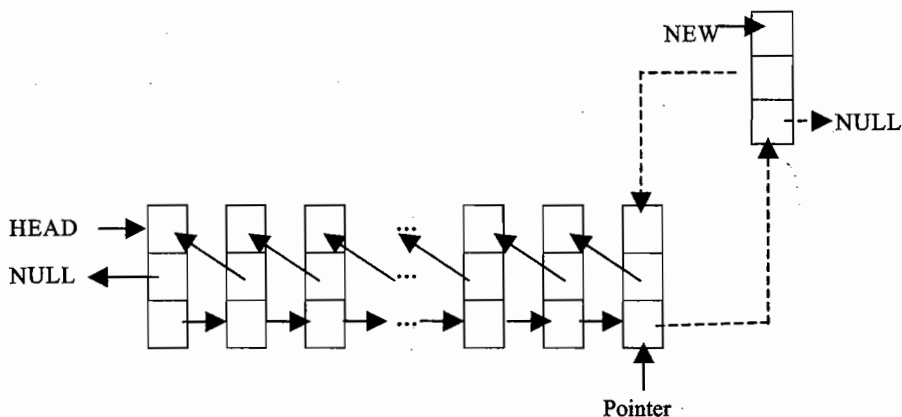


### 3. 插入在双链表尾端



新的节点插入在链表的尾端(Pointer 节点)。第一步, 必须将新节点的 Next 指针指向 Pointer 节点的指针(NULL)、第二步再将新节点的 Back 指针指向 Pointer 节点, 第 3 步将 Pointer 的 Next 指针指向新节点, 最后再将新节点 Next 指针的 Back 指针指向新节点。

```
NEW->Next = Pointer->Next
NEW->Back = Pointer
Pointer->Next = NEW
```

**程序实例:**

设计一个双链表内节点插入的程序。

**程序构思:**

声明一个新节点(New)供用户输入欲插入节点的内容。

用户输入一个节点内容(Key), 表示欲插入在那一个节点之后。

持续往下一个节点, 直到节点内容等于 Key 或节点指针为 NULL(即找不到该节点)。

如果该节点不存在, 则插入在首节点前:

```
NEW->Next = Pointer
Pointer->Back = New
HEAD = New
```

如果找到该节点, 且在双链表中间:

```
NEW->Next = Pointer->Next
NEW->Back = Pointer
Pointer->Next = NEW
New->Next->Back = New
```

如果找到该节点, 且在双链表的尾端:

```
NEW->Next = Pointer->Next
NEW->Back = Pointer
Pointer->Next = NEW
```

**程序源代码:**

```
01 /* ===== Program Description ===== */
02 /* 程序名称: d_insert.c */
03 /* 程序目的: 设计一个双链表内节点插入的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define Max1 10
08 struct DList /* 节点结构声明 */
09 {
10 int Number;
```

```

11 struct DList *Back;
12 struct DList *Next;
13 };
14 typedef struct DList DNode;
15 typedef DNode *DLink;
16
17 int Data1[Max1] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
18
19 /* ----- */
20 /* 释放双链表 */
21 /* ----- */
22 void Free_DList(DLink Head)
23 {
24 DLink Pointer; /* 节点声明 */
25
26 while (Head != NULL) /* 当节点为 NULL 结束循环 */
27 {
28 Pointer = Head;
29 Head = Head->Next; /* 往下一个节点 */
30 free(Pointer);
31 }
32 }
33
34 /* ----- */
35 /* 插入节点至双链表内 */
36 /* ----- */
37 DLink Insert_DList(DLink Head,DLink New,int Key)
38 {
39 DLink Pointer; /* 节点声明 */
40
41 Pointer = Head; /* Pointer 指针设为首节点 */
42
43 while (1)
44 {
45 /* 插入在链表尾端 */
46 if (Pointer->Number == Key)
47 {
48 New->Next = Pointer->Next;
49 New->Back = Pointer;
50 Pointer->Next = New;
51 /* 插入在链表中间 */
52 if (New->Next != NULL)
53 New->Next->Back = New;
54 break;
55 }
56 /* 找不到数据, 插入在链表首节点前 */
57 if (Pointer->Number != Key && Pointer->Next == Head)
58 {
59 New->Next = Pointer;
60 Pointer->Back = New;
61 Head = New;
62 break;
63 }
64 Pointer = Pointer->Next; /* 往下一个节点 */
65 }
66 return Head;
67 }
68
69 /* ----- */
70 /* 输出双链表数据 */
71 /* ----- */

```



```

72 void Print_DList(DLink Head)
73 {
74 DLink Pointer; /* 节点声明 */
75 Pointer = Head; /* Pointer 指针设为首节点 */
76
77 printf("Input Data : \n");
78 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
79 {
80 printf("[%d]", Pointer->Number);
81 Pointer = Pointer->Next; /* 往下一个节点 */
82 }
83 printf("\n");
84 }
85
86 /* ----- */
87 /* 建立双链表 */
88 /* ----- */
89 DLink Create_DList(DLink Head,int *Data,int Max)
90 {
91 DLink New; /* 节点声明 */
92 DLink Pointer; /* 节点声明 */
93 int i;
94
95 Head = (DLink) malloc(sizeof(DNode)); /* 内存配置 */
96
97 if (Head == NULL) /* 内存配置失败 */
98 printf("Memory allocate Failure!!\n");
99 else
100 {
101 Head->Number = Data[0]; /* 定义首节点数据编号 */
102 Head->Back = NULL;
103 Head->Next = NULL;
104
105 Pointer = Head; /* Pointer 指针设为首节点 */
106
107 for (i=1;i<Max;i++)
108 {
109 /* 内存配置 */
110 New = (DLink) malloc(sizeof(DNode));
111
112 New->Number = Data[i];
113 New->Back = NULL;
114 New->Next = NULL;
115 /* 将新节点串连在原列表尾端 */
116 Pointer->Next = New;
117 /* 新节点的前一个节点为原列表尾端 */
118 New->Back = Pointer;
119 /* 列表尾端节点为新节点 */
120 Pointer = New;
121 }
122 }
123 return Head;
124 }
125
126 /* ----- */
127 /* 主程序 */
128 /* ----- */
129 void main ()
130 {
131 DLink Head; /* 节点声明 */
132 DLink New;

```

```

133 int Key;
134
135 Head = Create_DList(Head, Data1, Max1); /*调用建立循环链表*/
136
137 if (Head != NULL)
138 {
139 Print_DList(Head); /* 调用输出循环链表数据 */
140 while (1)
141 {
142 printf("Input 0 to EXIT\n"); /* 数据输入提示 */
143 New = (DLink) malloc(sizeof(DNode)); /* 内存配置 */
144 printf("Please input the data : ");
145 scanf("%d", &New->Number);
146 if (New->Number == 0) /* 输入 0 时结束循环 */
147 break;
148 printf("Please input the data number for Insert : ");
149 scanf("%d", &Key);
150
151 Head = Insert_DList(Head, New, Key); /* 调用插入节点 */
152 Print_DList(Head); /* 输出链表数据 */
153 }
154 Free_DList(Head);
155 }
156 }

```

运行结果:

```

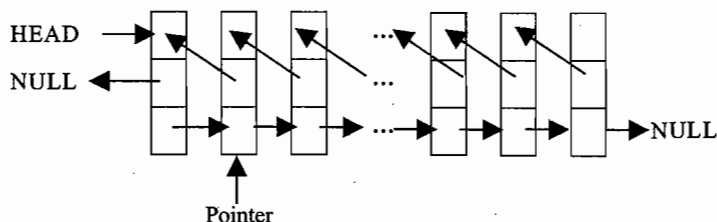
C:\DS>d_insert
Input Data :
[1][2][3][4][5][6][7][8][9][10]
Input 0 to EXIT
Please input the data : 60
Please input the data number for Insert : 1
Input Data :
[1][60][2][3][4][5][6][7][8][9][10]
Input 0 to EXIT
Please input the data : 70
Please input the data number for Insert : 5
Input Data :
[1][60][2][3][4][5][70][6][7][8][9][10]
Input 0 to EXIT
Please input the data : 80
Please input the data number for Insert : 10
Input Data :
[1][60][2][3][4][5][70][6][7][8][9][10][80]
Input 0 to EXIT
Please input the data : 0
C:\DS>

```

### 10.2.3 双链表的删除

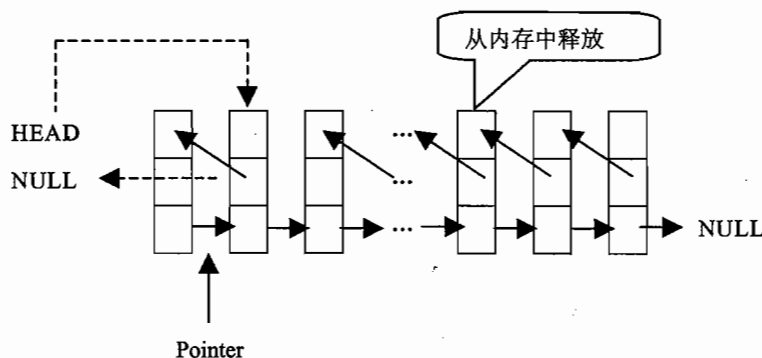
双链表内节点的删除, 依欲删除节点的位置来分, 可分为下列几类:

## 1. 删除双链表首节点:

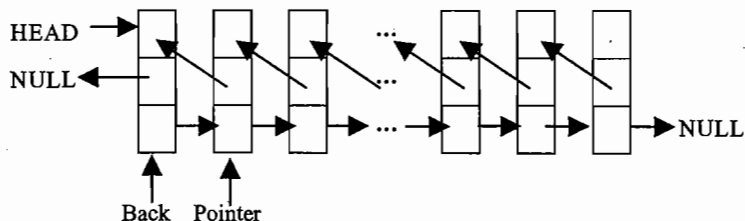


删除的节点在链表的开头, 需要将首节点指向首节点的指针(即下一个节点), 并将首节点的 Back 指针指向 NULL, 最后把原来的节点从内存中释放。

```
HEAD = Pointer->Next
HEAD->Back = NULL
Free(Pointer)
```

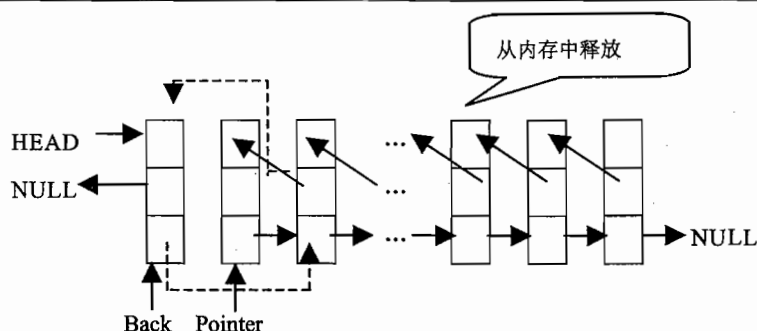


## 2. 删除双链表中间节点

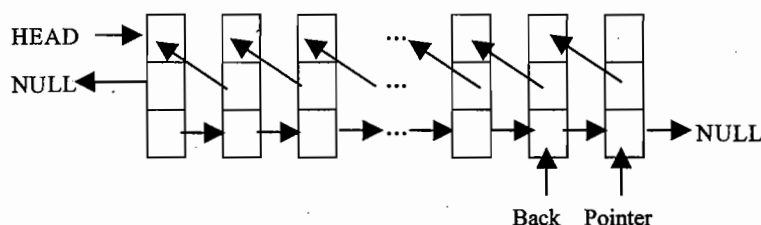


删除的节点在链表的中间, 如果我们找到 Pointer 节点, 则需要将前一个节点的指针指向 Pointer 节点的指针(即下一个节点), 再将下一个节点的 Back 指针指向前一个节点, 最后将原来的节点从内存中释放。

```
Back = Pointer->Back
Back->Next = Pointer->Next
Pointer->Next->Back = Back
Free(Pointer)
```

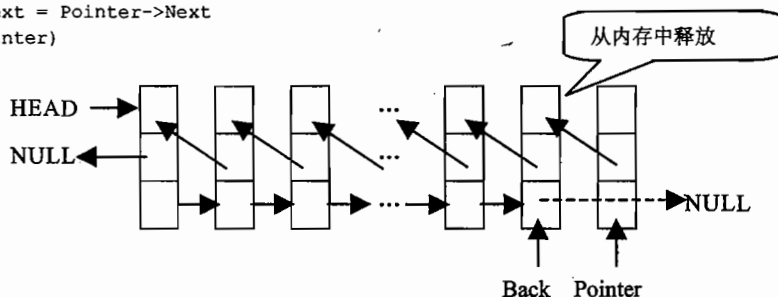


### 3. 删除双链表尾端的节点:



删除的节点在链表的尾端(Pointer 节点), 如果我们找到 Pointer 节点, 则需要将前一个节点的指针指向 Pointer 节点的指针(NULL)。最后将原来的节点从内存中释放。

```
Back = Pointer->Back
Back->Next = Pointer->Next
Free(Pointer)
```



### 程序实例:

设计一个删除双链表内节点的程序。

### 程序构思:

持续往下一个节点查找欲删除节点, 直到节点内容找到或节点指针为 NULL(即找不到该节点)。

如果该节点不存在, 输出消息说节点不存在。

如果该节点存在, 且是首节点:

```
HEAD = Pointer->Next
HEAD->Back = NULL
Free(Pointer)
```

如果该节点存在, 且是双向链连列表中间节点, 则:

```
Back = Pointer->Back
Back->Next = Pointer->Next
```

```

 Pointer->Next->Back = Back
 Free(Pointer)

```

如果该节点存在，且是双向链连列表的尾端，则：

```

Back = Pointer->Back
Back->Next = Pointer->Next
Free(Pointer)

```

程序源代码：

```

01 /* ===== Program Description ===== */
02 /* 程序名称: d_insert.c */
03 /* 程序目的: 设计一个双链表内节点插入的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define Max1 10
08 struct DList /* 节点结构声明 */
09 {
10 int Number;
11 struct DList *Back;
12 struct DList *Next;
13 };
14 typedef struct DList DNode;
15 typedef DNode *DLink;
16
17 int Data1[Max1] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
18
19 /* ----- */
20 /* 释放双链表 */
21 /* ----- */
22 void Free_DList(DLink Head)
23 {
24 DLink Pointer; /* 节点声明 */
25
26 while (Head != NULL) /* 当节点为 NULL 结束循环 */
27 {
28 Pointer = Head;
29 Head = Head->Next; /* 往下一个节点 */
30 free(Pointer);
31 }
32 }
33
34 /* ----- */
35 /* 删除双链表内节点 */
36 /* ----- */
37 DLink Delete_DList(DLink Head,int Key)
38 {
39 DLink Pointer; /* 节点声明 */
40 DLink Back;
41
42 Pointer = Head; /* Pointer 指针设为首节点 */
43
44 while (1)
45 {
46 if (Pointer->Next == NULL)
47 {
48 printf("Not Found!!\n");
49 break;
50 }
51 if (Head->Number == Key) /* 删除首节点 */

```

```

52 {
53 Head = Pointer->Next;
54 Head->Back = NULL;
55 free(Pointer);
56 break;
57 }
58 Pointer = Pointer->Next; /* 往下一个节点 */
59 if (Pointer->Number == Key) /* 插入在链表尾端 */
60 {
61 Back = Pointer->Back;
62 Back->Next = Pointer->Next;
63 /* 插入在链表中间 */
64 if (Pointer->Next != NULL)
65 Pointer->Next->Back = Back;
66 free(Pointer);
67 break;
68 }
69 }
70 return Head;
71 }
72
73 /* ----- */
74 /* 输出双链表数据 */
75 /* ----- */
76 void Print_DList(DLink Head)
77 {
78 DLink Pointer; /* 节点声明 */
79 Pointer = Head; /* Pointer 指针设为首节点 */
80
81 printf("Input Data : \n");
82 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
83 {
84 printf("[%d]",Pointer->Number);
85 Pointer = Pointer->Next; /* 往下一个节点 */
86 }
87 printf("\n");
88 }
89
90 /* ----- */
91 /* 建立双链表 */
92 /* ----- */
93 DLink Create_DList(DLink Head,int *Data,int Max)
94 {
95 DLink New; /* 节点声明 */
96 DLink Pointer; /* 节点声明 */
97 int i;
98
99 Head = (DLink) malloc(sizeof(DNode)); /* 内存配置 */
100
101 if (Head == NULL) /* 内存配置失败 */
102 printf("Memory allocate Failure!!\n");
103 else
104 {
105 Head->Number = Data[0]; /* 定义首节点数据编号 */
106 Head->Back = NULL;
107 Head->Next = NULL;
108
109 Pointer = Head; /* Pointer 指针设为首节点 */
110
111 for (i=1;i<Max;i++)
112 {

```

```

113 /* 内存配置 */
114 New = (DLink) malloc(sizeof(DNode));
115
116 New->Number = Data[i];
117 New->Back = NULL;
118 New->Next = NULL;
119 /* 将新节点串连在原列表尾端 */
120 Pointer->Next = New;
121 /* 新节点的前一个节点为原列表尾端 */
122 New->Back = Pointer;
123 /* 列表尾端节点为新节点 */
124 Pointer = New;
125 }
126 }
127 return Head;
128 }
129
130 /* ----- */
131 /* 主程序 */
132 /* ----- */
133 void main ()
134 {
135 DLink Head; /* 节点声明 */
136 int Key;
137
138 Head = Create_DList(Head, Data1, Max1); /*调用建立循环链表*/
139
140 if (Head != NULL)
141 {
142 Print_DList(Head); /* 调用输出循环链表数据 */
143 while (1)
144 {
145 printf("Input 0 to EXIT\n"); /* 数据输入提示 */
146 printf("Please input the data number for Insert : ");
147 scanf("%d",&Key);
148 if (Key == 0) /* 输入 0 时结束循环 */
149 break;
150
151 Head = Delete_DList(Head,Key); /* 调用插入节点 */
152 Print_DList(Head); /* 输出链表数据 */
153 }
154 Free_DList(Head);
155 }
156 }

```

运行结果:

```

C:\DS>d_delete
Input Data :
[1][2][3][4][5][6][7][8][9][10]
Input 0 to EXIT
Please input the data number for Insert : 1
Input Data :
[2][3][4][5][6][7][8][9][10]
Input 0 to EXIT
Please input the data number for Insert : 6
Input Data :
[2][3][4][5][7][8][9][10]
Input 0 to EXIT
Please input the data number for Insert : 10

```

```

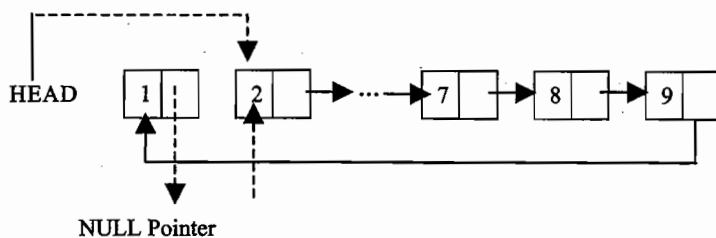
Input Data :
[2][3][4][5][7][8][9]
Input 0 to EXIT.
Please input the data number for Insert : 0
C:\DS>

```

## 【习题】

### 一、复习:

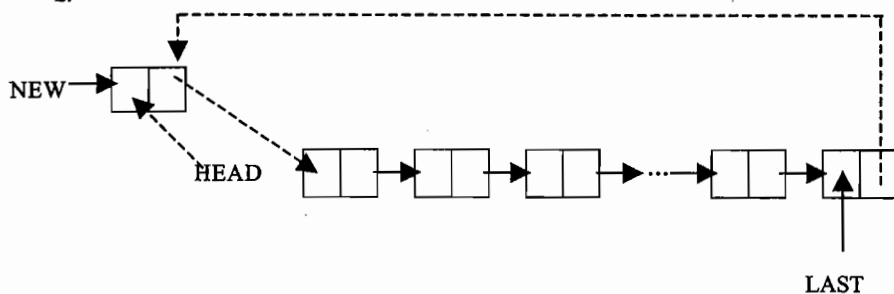
1.



上图所代表的程序语句为:

- (a) `Pointer = Head;`  
`Head->Next = NULL;`  
`Head = Pointer;`
- (b) `Pointer = Head->Next;`  
`Head->Next = NULL;`  
`Head = Pointer;`
- (c) `Head->Next = Pointer;`  
`Head = NULL;`
- (d) 以上皆非。

2.



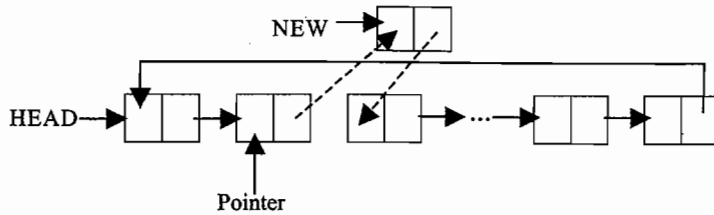
上图所代表的程序语句为:

- (a) `NEW->Next = HEAD`  
`LAST->Next = NEW`  
`HEAD = NEW`
- (b) `Head = New->Next;`  
`LAST = NEW;`  
`NEW = HEAD;`
- (c) `NEW = HEAD->Next;`  
`LAST = NEW;`  
`HEAD = NEW;`



(d) 以上皆非。

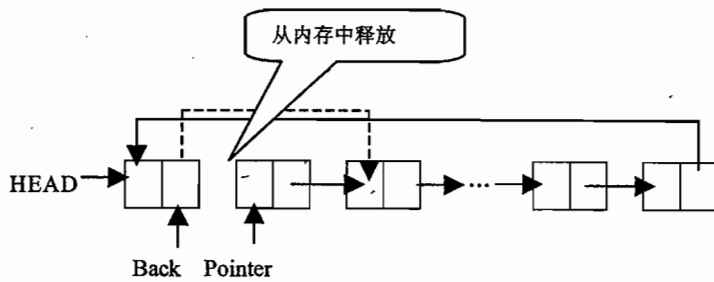
3.



上图所代表的程序语句为:

- (a) `NEW->Next = Pointer->Next;`  
`Pointer->Next = NEW;`
- (b) `Pointer->Next = NEW;`  
`NEW->Next = Pointer->Next;`
- (c) `NEW = Pointer->Next;`  
`Pointer = NEW;`
- (d) 以上皆非。

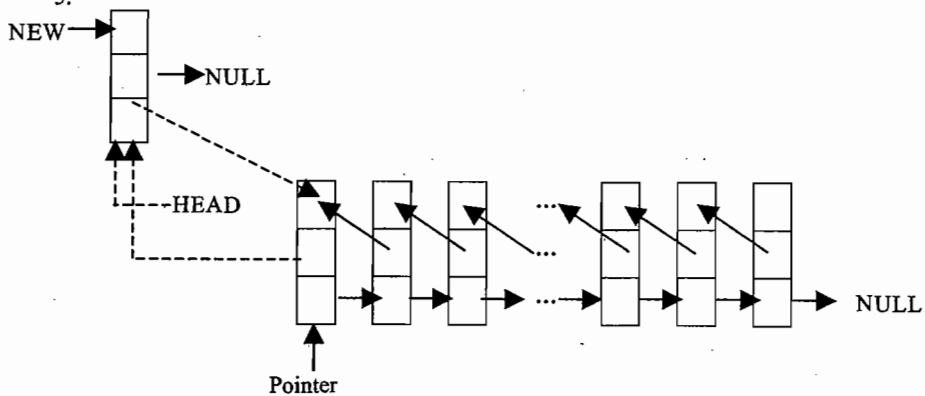
4.



上图所代表的程序语句为:

- (a) `Free(Pointer);`  
`Back->Next = Pointer->Next;`
- (b) `Free(Pointer);`  
`Back = Pointer;`
- (c) `Back->Next = Pointer->Next;`  
`Free(Pointer);`
- (d) 以上皆非。

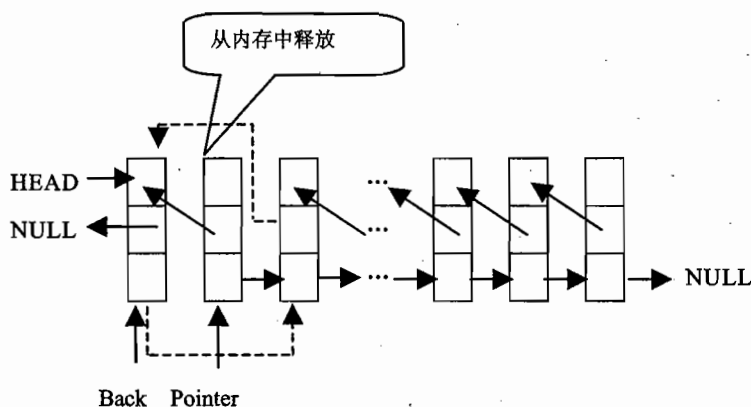
5.



上图所代表的程序语句为:

- (a) `NEW->Back = Pointer;`  
`Pointer->Next = New;`  
`HEAD = New;`
- (b) `NEW->Next = Pointer;`  
`Pointer->Back = New;`  
`HEAD = New;`
- (c) `NEW = Pointer;`  
`HEAD = New;`
- (d) 以上皆非。

6.



上图所代表的程序语句为:

- (a) `Pointer->Back = HEAD;`  
`Back->Next = Pointer;`  
`Free(Pointer);`
- (b) `Pointer = HEAD;`  
`Back = Pointer;`  
`Free(Pointer);`
- (c) `Back = Pointer->Back;`  
`Back->Next = Pointer->Next;`  
`Pointer->Next->Back = Back;`  
`Free(Pointer);`
- (d) 以上皆非。

二、应用:

1. 试写出将两个循环链表的相连的子程序。
2. 试写出复制循环链表的子程序。
3. 试写出建立双向循环链表的子程序。
4. 试写出从双链表尾端印数据到双链表首节点子程序。



# 字符串结构

## 第 11 章

- ◆ 字符串的声明
- ◆ 字符串的基本 I/O
- ◆ 字符串的传递方式
- ◆ 字符串的基本处理
- ◆ 字符串的高级处理
- ◆ 字符串转换数值的应用

字符串(String)是一连串的字符集,例如:“Thomas is smart”即为一个字符串。C语言在提供各种数值(整数、浮点数等)运算之外,也提供了字符串的处理,使程序语言在与人的连接上更具亲和力。

字符串是一个结构性的数据类型,其表示的方式有两种,一种是数组结构,另一种是使用指针,而C语言两种方法都有提供。本章将详细介绍字符串的声明、I/O及传递方式,并讨论字符串的各种处理方式及转换数值的应用。

## 11.1 字符串的声明

C语言的字符串可用字符数组及指针两种方式来表示。若以字符数组表示,其最后必须以空字符“\0”结束,且该空字符也算字符串中的一个字符。例如:“I am happy!”中有11个字符,但其后还隐含了一个空字符,故存储字符串为12个字符。

字符串的声明有下列3种方式:

### (1) 使用数组—设置字符串长度

#### (a) 不设置字符串初值

```
char 字符串变量[字符串长度];
ex: char S1[20];
```

#### (b) 设置字符串初值

```
char 字符串变量[字符串长度] = “字符串常数”;
ex: char S1[12] = “I am happy!”;
```

### (2) 使用数组—不设置字符串长度

```
char 字符串变量[字符串长度] = “字符串常数”;
ex: char S1[] = “always happy!”;
```

### (3) 使用指针变量

```
char 字符串变量[字符串长度] = “字符串常数”;
ex: char *S1= “happy forever”;
```

第一种方式是以数组结构来存储字符串,可以在声明的同时设置初值,只要在字符串变量后面加上一个字符串常数(字符串常数前后要加上双引号“”),即可设置字符串初值。另外要特别注意,字符串的结尾处一定要有一个结束字符“\0”,所以声明的长度至少要比实际存入数组的字符串长度多1。如例子中要将“I am happy!”这11个字符存入字符串数组S1中,那么声明的长度至少要大于等于12才可以,否则在编译时会出现错误消息。若未设置字符串初始值,可利用字符串函数来设置字符串的内容。

```
char S1[12]=“I am happy”;
```

|   |   |   |   |   |   |   |   |   |   |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| I |   | a | m |   | h | a | p | p | y | !  | \0 |

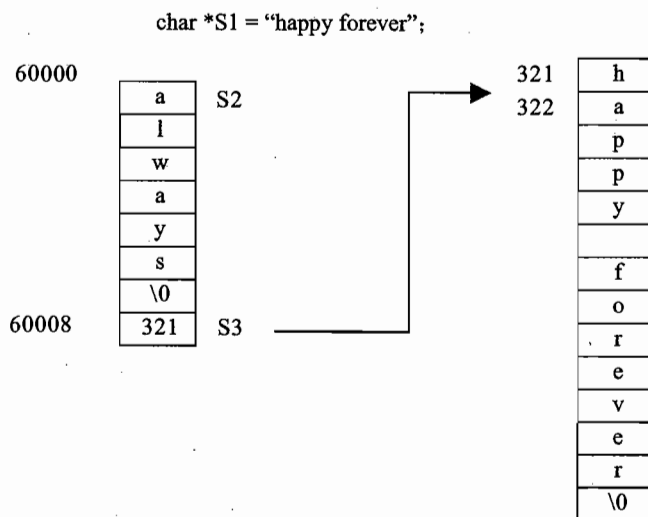
↑  
字符串结束字符

第2种方式也是以数组结构存储字符串,在声明的同时不指定数组长度,但当字符串初值设置后,系统会按照字符串初值的长度去决定字符串数组的长度,并自动在结尾处加上一个结束字符“\0”。

```
char S1[] = “always happy”;
```

|   |   |   |   |   |   |  |   |   |   |   |   |    |
|---|---|---|---|---|---|--|---|---|---|---|---|----|
| A | l | w | a | y | s |  | h | a | p | p | y | \0 |
|---|---|---|---|---|---|--|---|---|---|---|---|----|

第3种方式是用一个指针变量来作为字符串变量,将字符串存到所配置的空间,再由指针指到所配置空间的起始位置。可以说“字符串变量名称就是一个指针”。不论是用那一种方式,字符串的最大长度均为65535个字符。



程序源代码:

```

01 /* ===== Program Description===== */
02 /* 程序名称: String01.c */
03 /* 程序目的: 字符串声明的 3 种方式 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 main()
07 {
08 char S1[7] = "Thomas" ;
09 char S2[] = "is" ;
10 char *S3 = "smart" ;
11 printf ("\n %s %s %s", S1,S2,S3);
12 }

```

运行结果:

```

C:\DS>String01
Thomas is smart
C:\DS>

```

## 11.2 字符串的基本 I/O

字符串的输入与输出各有两种方法。第 1 种输入字符串的方法可使用 scanf( ) 含数，在输入格式中要设置 "%s"，再加上字符串变量名称。

例如:

```

char name[20];
printf("Input your name : ");
scanf("%s",name);

```

第 2 种输入字符串的方法是使用 get( ) 函数，格式为: `get(字符串变量)`

例如:

```

char name[20];
printf("Input your name : ");
get(name);

```

此两种输入方式的不同在于使用 scanf( ) 时，字符串中不可以含有空的字符。而 get( ) 可输入含空格的字符串。

输出字符串也有两种方式,第1种是使用 printf() 函数,和 scanf() 一样要设置输出格式“%s”,再加上字符串变量名称。

例如: `printf("Your name is %s",name);`

第2种输出字符串的方式是使用 puts() 函数,格式为:

`puts(字符串变量);`

例如: `printf("Your name is ");`  
`puts(name);`

程序源代码:

```
01 /* ===== Program Description===== */
02 /* 程序名称: String02.c */
03 /* 程序目的: 字符串的基本 I/O(1)---使用 scanf(), printf() */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 main()
07 {
08 char color[20];
09 printf("\n What is your favorite color ?");
10 scanf("%s", color); /*读取输入字符串*/
11 printf("\n Your favorite color is %s ", color); /*输出字符串*/
12 }
```

运行结果:

```
C:\DS>String02
What is your favorite color ? Red
Your favorite color is Red.

C:\DS>
```

程序源代码:

```
01 /* ===== Program Description===== */
02 /* 程序名称: String03.c */
03 /* 程序目的: 字符串的基本 I/O(2)---使用 gets(), puts() */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 main()
07 {
08 char feeling[20];
09 printf("\n How about this clothes ? ");
10 gets(feeling);
11 printf("My feeling is ");
12 puts(feeling);
13 }
```

运行结果:

```
C:\DS>String03
How about this clothes? Beautiful
My feeling is beautiful.

C:\DS>
```

## 11.3 字符串的传递方式

字符串在函数间是以传址调用的方式传递,所以在函数中对于字符串的修改都会直接影响到主程序中原来的字符串内容。因为字符串变量名称就是指针,故传递的参数即为字符串名称。

```
main()
{
```

```

 void print();
 char book[] = "Data Structure";
 print(book);
 }

 void print(char s[]) {
 puts(s);
 }

```

其中 `print( )` 函数为用来打印字符串，必须将字符串传入 `print( )` 中，由于字符串变量“book”本身即为指针，故传递的参数即为字符串名称“book”。

程序源代码：

```

01 /* ===== Program Description===== */
02 /* 程序名称: String04.c */
03 /* 程序目的: 字符串的传递 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 main ()
07 {
08 void Input_String(), Output_String();
09 char S1[30];
10 printf("\n Input string :");
11 Input_String(S1);
12 Output_String(S1);
13 }
14 void Input_String(char s) /*子程序:字符串输入*/
15 {
16 gets(s);
17 }
18 void Output_String(char s[]) /*子程序:字符串输出*/
19 {
20 printf("\n Output string: ");
21 puts(s);
22 }

```

运行结果：

```

C:\DS>String04
Input string: I am happy.
Output string: I am happy.

C:\DS>

```

将 `Input string` 存入字符串变量“S1”，再把字符串变量名称“S1”当做参数传给 `Output_String( )` 进行输出。其中参数传递用 `*s` 或 `s[ ]` 两种方式均可。

## 11.4 字符串的基本处理

C 语言对于字符串的操作提供了许多字符串处理函数，本章将针对字符串的基本处理来说明这些函数所应用的技巧。以下各节的程序将统一采用“字符数组结构”来处理字符串的操作。

字符串的基本处理有下列几种：

1. 字符串的长度计算
2. 字符串的复制
3. 字符串的结合



4. 字符串的取代
5. 字符串的插入
6. 字符串的删除

以下各小节将对这几种字符串基本的处理做更详细的说明。

### 11.4.1 字符串的长度计算: Strlen(char \*s)

Strlen( ) 是用来计算字符串的长度, 其中并不包含结束字符。此函数会计算字符串的字节长, 不把结束字符算在内。若以数组结构来声明字符串, 声明的字符串长度是表示分配给该字符串的内存空间大小, Strlen()所回传的字符串长度并非声明的字符串长度, 而是结束字符“\0”之前的字符数。

例如:

```
char name[50] = "Thomas";
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6  |
|---|---|---|---|---|---|----|
| T | h | o | m | a | s | \0 |

字符串的长度为 6 而非 50。由此可知, 空字符串的长度为 0。

程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: String05.c */
03 /* 程序目的: 计算字符串的长度 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 计算字符串的长度 */
08 /*-----*/
09 int Strlen(char *s)
10 {
11 int i;
12 for (i=0; s[i]!='\0';) /*用循环来计算字符串长度*/
13 i++;
14 return i; /*将累加的 i 值返回给 Strlen()*/
15 }
16 /*-----*/
17 /* 主程序: 读取字符串长度后, 输出该字符串的长度 */
18 /*-----*/
19 void main()
20 {
21 char string[50]; /*声明长度为 50 的字符串数组*/
22 int length; /*字符串长度*/
23
24 printf("\n Please input string : ");
25 gets(string); /*读取字符串存到变量 string*/
26 length=Strlen(string); /*计算字符串 string 的长度*/
27 printf("\n The input string length is %d ",length);
28 }
```

运行结果:

```
C:\DS>String05
Please input string : How are you ?
The input string length is 12

C:\DS>
```

### 11.4.2 字符串的复制——Strcpy(char \*s1, char \*s2)

Strcpy() 是将一个字符串复制到另一个字符串。若要将字符串 s2 复制到字符串 s1, 则 s1 必须要有足够的空间来容纳 s2, 最后此函数会返回 s1 的起始地址。字符串复制的用途为可将字符串内容备份起来, 以免在进行字符串的其它处理时更改了字符串的原始内容。

例如:

|    |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|----|----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
|    | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| S1 | \0 |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |

|    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
|    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| S2 | T | h | o | m | a | s |   | i | s |   | s  | m  | a  | r  | t  | \0 |

→ 将 S2 复制到 S1

|    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
|    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| S1 | T | h | o | m | a | s |   | i | s |   | s  | m  | a  | r  | t  | \0 |

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: String06.c */
03 /* 程序目的: 字符串的复制 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 将字符串 S2 复制到字符串 S1 */
08 /*-----*/
09 char *Strcpy(char *s1, char *s2)
10 {
11 int i;
12
13 for (i=0; s2[i]!='\0'; i++)
14 s1[i] = s2[i]; /*逐一复制字符串的内容*/
15 s1[i]='\0'; /*设置字符串结束*/
16 return s1; /*将字符串 s1 返回给 Strcpy()*/
17 }
18
19 /*-----*/
20 /* 主程序: 读取字符串存入 string 后, 复制到字符串 copysting */
21 /*-----*/
22 void main()
23 {
24 char string [50]; /*声明原始字符串*/
25 char copysting[50]; /*声明复制字符串*/
26
27 printf("\n Please input string : ");
28 gets(string); /*读取字符串存到变量 string*/
29 Strcpy(copysting, string); /*将字符串 string 复制到字符串 copysting*/
30
31 printf("\n String : %s ", string); /*输出原始字符串*/
32 printf("\n Copysting : %s", copysting); /*输出复制字符串*/
33 }

```

运行结果:

```
C:\DS>String06
Please input string : How are you ?

String : How are you ?
Copystring : How are you ?

C:\DS>
```

### 11.4.3 字符串的结合——Strcat(char \*s1, char \*s2)

字符串的结合是将两个字符串合并成一个字符串。若要将字符串 S2 连结到字符串 S1 的后面,则必须将 S1 的结束字符去掉。且字符串 S1 要有足够的空间来容纳字符串 S2,最后此函数会返回 S1 的起始地址。

例如:

```
char S1[20] = "Thomas ";
char S2[20] = "is smart ";
```

|    |   |   |   |   |   |   |   |    |  |
|----|---|---|---|---|---|---|---|----|--|
|    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  |  |
| S1 | T | h | o | m | a | s |   | \0 |  |

|    |   |   |   |   |   |   |   |   |    |
|----|---|---|---|---|---|---|---|---|----|
|    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8  |
| S1 | i | s |   | s | m | a | r | t | \0 |

→将 S2 结合到 S1

|    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
|    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| S1 | T | h | o | m | a | s |   | i | s |   | s  | m  | a  | r  | t  | \0 |

程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: String07.c */
03 /* 程序目的: 字符串的结合 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 将字符串 s2 合并到字符串 s1 的后面 */
08 /*-----*/
09 char *Strcat(char *s1, char *s2)
10 {
11 int i, j;
12
13 for (i=0; s1[i] != '\0'; i++); /*寻找 s1 的结束位置*/
14 for (j=0; s2[j] != '\0'; j++) /*复制字符串的内容*/
15 s1[i+j]=s2[j];
16 s1[i+j]='\0'; /*设置合并后的字符串结束*/
17 return s1; /*将 s1 返回给 Strcat()*/
18 }
19 /*-----*/
20 /* 主程序:读入两个字符串后, 将其合并成一个字符串 */
21 /*-----*/
22 void main()
23 {
```

```

24 char string1 [100]; /*声明字符串数组 1*/
25 char string2 [50]; /*声明字符串数组 2*/
26
27 printf("\n Please input string (1):");
28 gets(string1); /*读取字符串 1 存到变量 string1*/
29 printf("\n Please input string (2):");
30 gets(string2); /*读取字符串 2 存到变量 string2*/
31
32 printf("\n String1:%s",string1);
33 printf("\n String2:%s",string2);
34
35 Strcat(string1,string2); /*结合两字符串*/
36
37 printf("\nThe merge string is : %s ",string1); /*输出合并后的字符串*/
38 }

```

运行结果:

```

C:\DS>String07
Please input string (1): Thomas
Please input string (2): is smart

String1: Thomas
String2: is smart
The merge string is : Thomas is smart

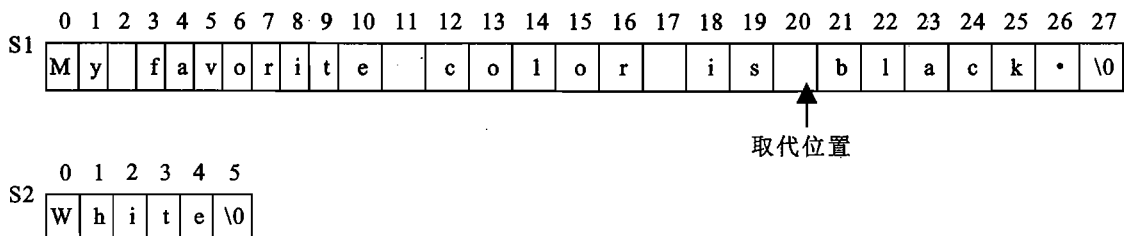
C:\DS>

```

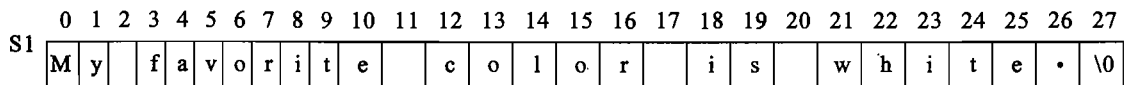
#### 11.4.4 字符串的取代——Strrep(char \*s1, char \*s2, int pos)

字符串的取代是将字符串中的某一子字符串用另一个子字符串的内容来替换。需特别注意的是,这种字符串处理的方法并不会改变字符串的长度,若替换的子字符串较原子字符串长,则会覆盖掉原字符串中的内容。

例如:



取代后:



子字符串 S2“white”是从原字符串 S1 的索引值“21”开始取代 5 个字符,但真正取代的位置是计算其在原字符串中的第几个位置,而非索引值,故上例的取代位置为 22 而非 21。

## 程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: String08.c */
03 /* 程序目的: 字符串的取代 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 将字符串 s2 从字符串 s1 的某一个位置 pos 取代 s1 的子串 */
08 /*-----*/
09 char *Strrep(char *s1, char *s2, int pos)
10 {
11 int i, j;
12
13 pos--; /*计算取代的起始位置*/
14 i=0;
15 for (j=pos; s1[j]!='\0'; j++) /*从取代的起始位置开始*/
16 if (s2[i] != '\0')
17 {
18 s1[j]=s2[i]; /*进行取代*/
19 i++;
20 } else {
21 break;
22 }
23 return s1;
24 }
25 /*-----*/
26 /* 主程序:输入两字符串,再进行字符串取代 */
27 /*-----*/
28 void main()
29 {
30 char string1 [100]; /*声明字符串数组 1*/
31 char string2 [50]; /*声明字符串数组 2*/
32 int position; /*进行取代的起始位置*/
33
34 printf("\nPlease input original string: "); /*读取原始字符串并存入 "String1"*/
35
36 gets(string1);
37 printf("\nPlease input substitute string: "); /*读取欲替换的子字符串并存入 "String
2"*/
38 gets(string2);
39 printf("\nPlease input substitute position: "); /*读取进行字符串替换的起始位置*/
40
41 scanf("%d", &position);
42 Strrep(string1, string2, position); /*进行字符串的取代*/
43 printf("\nThe final string :%s \n", string1);
44 }

```

## 运行结果:

```

C:\DS>String08
Please input substitute string: My favorite color is black.
Please input substitute string: white
Please input substitute position: 22
The final string : My favorite color is white.

C:\DS>

```

### 11.4.5 字符串的插入——Strins(char \*s1, char \*s2, int pos)

将欲插入的字符串插入至原字符串中的某个位置，插入完成后，字符串的总长度会是两个字符串的总合。

例如：

原字符串——S1

长度：13

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|
| I |   | a | m |   | a |   | g | i | r | l  | .  | \0 |

欲插入的字符串——S2（其后需加一空格符，以和原字符串的字符隔开）

长度：6

| 0 | 1 | 2 | 3 | 4 | 5  |
|---|---|---|---|---|----|
| h | a | p | p | y | \0 |

步骤 1：先挪出欲插入字符串的空间

长度：19

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|
| I |   | a | m |   | a |   |   |   |   |    |    |    | g  | i  | r  | l  | .  | \0 |

空出 6 个字符空间

步骤 2：插入欲插入的字符串

长度：19

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|
| I |   | a | m |   | a |   | h | a | p | p  | y  |    | g  | i  | r  | l  | .  | \0 |

程序源代码：

```
01 /* ===== Program Description ===== */
02 /* 程序名称: String09.c */
03 /* 程序目的: 字符串的插入 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 计算字符串的长度 */
08 /*-----*/
09 int Strlen(char *s)
10 {
11 int i;
12
13 /*若不为结束字符则字符串长度加 1*/
14 for (i=0; s[i]!='\0';) /*用循环来计算字符串长度*/
15 i++;
16 return i; /*将累加的 i 值返回给 Strlen() */
17 }
18
19 /*-----*/
20 /* 进行字符串的插入 */
21 /*-----*/
22 char *Strins(char *s1, char *s2, int pos)
```

```

23 {
24 int s1_length; /*字符串"s1"之长度*/
25 int s2_length; /*字符串"s2"之长度*/
26 int i,j;
27
28 pos--; /*字符串的开始位置*/
29
30 s1_length = Strlen(s1); /*计算字符串"s1"的长度存入 s1_length */
31 s2_length = Strlen(s2); /*计算字符串"s2"的长度存入 s2_length */
32
33 /*将欲插入之位置之后的所有字符往后移,以空出 s2 所需的字符串长度*/
34 for (i=s1_length; i>=pos; i--)
35 s1[i+s2_length] = s1[i];
36
37 /*将字符串"s2"的内容填入所空出的字符串"s1"中*/
38 for (j=pos; s2[j-pos] !='\0';j++)
39 s1[j] = s2[j-pos];
40 return s1; /*将"s1"返回值给 Strins()*/
41 }
42
43 /*-----*/
44 /* 主程序:输入原始字符串.插入字符串及插入位置后输出插入结果 */
45 /*-----*/
46 void main()
47 {
48 char string1[100]; /*声明字符串数组(1)*/
49 char string2[50]; /*声明字符串数组(2)*/
50 int position; /*欲插入字符串之位置*/
51
52 /*读取原始字符串并存入"string1"*/
53 printf("\nPlease input original string:");
54 gets(string1);
55 /*读取插入字符串并存入"string1"*/
56 printf("Insertion string(add an empty char):");
57 gets(string2);
58 /*读取插入的位置并存入"position"*/
59 printf("Insertion position:");
60 scanf("%d",&position);
61 /*进行字符串插入*/
62 Strins(string1,string2,position);
63 printf("\nThe final string is: %s \n",string1);
64 }

```

运行结果:

```

C:\DS>String09
Please input original string: Thomas is a boy
Insertion string(add an empty char):smart
Insertion position:13

The final string is: Thomas is a smart boy

C:\DS>

```

#### 11.4.6 字符串的删除——Strdel(char \*s1, int pos, int len)

字符串的删除 Strdel() 是从 S1 某个特定的位置 pos 开始, 依某长度 len 删除子字符串。

例如:

原字符串——S1

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| Y | o | u |   | A | r | e |   | a |   | s  | m  | a  | r  | T  |    | b  | o  | y  | .  | \0 |    |

删除起始位置 pos

删除后(删除长度: 6)

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| Y | o | u |   | A | r | e |   | a |   | b  | o  | y  | .  | \0 |

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: String10.c */
03 /* 程序目的: 字符串的删除 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*----- */
07 /* 从字符串 S1 的某个位置 pos 删除长度 len 的子字符串 */
08 /*----- */
09 char *Strdel(char *s,int pos, int len)
10 {
11 int i;
12 pos--; /*计算取代的起始位置*/
13
14 for (i=pos +len; s[i]!='\0';i++) /*从 pos 开始删除字符串*/
15 s[i-len]=s[i];
16 s[i-len]='\0';
17 return s;
18 }
19
20 /*----- */
21 /* 主程序:读入字符串及删除信息,进行删除后打印删除结果 */
22 /*----- */
23 void main()
24 {
25 char string[50]; /*声明字符串数组*/
26 int position; /*删除起始位置*/
27 int length; /*欲删除的长度*/
28
29 printf("\nPlease input original string: ");/*读取原始字符串,并存入"String1"*/
30 gets(string);
31 printf("\nPlease input delete position: ");/*读取欲删除的起始位置*/
32 scanf("%d",&position);
33 printf("\nPlease input delete length: "); /*读取欲删除字符串的长度*/
34 scanf("%d",&length);
35 Strdel(string,position,length); /*删除字符串*/
36 printf("\nThe final string : %s",string);
37 }

```

运行结果:

```

C:\DS>String10
Please input original string: You are a smart boy.
Please input delete position: 11
Please input delete length: 6
The final string : You are a boy.

```



C:\DS&gt;

## 11.5 字符串的高级处理

在 11.4 中介绍了字符串的基本操作处理, 本节将进一步说明较复杂的字符串处理功能。

本节将介绍的字符串高级处理有下列几种:

1. 字符串的比较
2. 抽取子字符串
3. 字符串的比较
4. 字符串的分割
5. 常用的字符串函数

### 11.5.1 字符串的比较——Strcmp(char \*s1, char \*s2)

字符串的比较是针对两字符串的每个字符两两做比较(大小写视为不同), 直到两字符串中有字符不相等或字符串已结束。若两字符串相等, 则返回值为 0。若两字符串不相等, 则用 ASCII 码来比较不同的字符, 以决定字符串的大小。

例如:

|    |   |   |   |   |   |   |   |   |   |   |    |    |
|----|---|---|---|---|---|---|---|---|---|---|----|----|
| S1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|    | I |   | a | m | h | a | p | p | y | . |    | \0 |

|    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |
|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| S2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|    | I |   | a | m | v | e | r | y |   | h | a  | p  | p  | Y  | .  |    | \0 |

字符串“S1”和字符串“S2”不相等, 则比较其不同的字符:

$S1[5] = 'h' < S2[5] = 'v'$

故字符串“S1” < 字符串“S2”

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: String11.c */
03 /* 程序目的: 字符串的比较 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 进行字符串的比较 */
08 /*-----*/
09 int Strcmp(char *s1, char *s2)
10 {
11 int i, j;
12
13 for (i=0; s1[i]==s2[i]; i++)
14 if (s1[i] == '\0' && s2[i] == '\0') /*字符串"s1"等于字符串"s2"*/
15 return 0; /*s1=s2 则返回值 0 给 Strcmp*/

```

```

16 if (s1[i] > s2[i]) /*字符串"s1"大于字符串"s2"*/
17 return 1; /*s1>s2 则返回值 1 给 Strcmp*/
18 return -1; /*s1<s2 则返回值-1 给 Strcmp*/
19 }
20 /*-----*/
21 /* 主程序:输入两个字符串后,用 Strcmp 比较两字符串的大小 */
22 /*-----*/
23 void main ()
24 {
25 char s1[50]; /*声明字符串数组"s1"*/
26 char s2[50]; /*声明字符串数组"s2"*/
27 int compare; /*存储比较结果的变量*/
28
29 printf("\nPlease input string (1) :");
30 gets(s1); /*读取字符串 string1*/
31 printf("\nPlease input string (2) :");
32 gets(s2); /*读取字符串 string2*/
33 compare=Strcmp(s1,s2); /*进行字符串的大小比较*/
34
35 printf("\nString(1):%s",s1);
36 printf("\nString(2):%s",s2);
37 printf("\nCompare result:");
38 switch (compare)
39 {
40 case 0: printf("\nString(1) = String(2)\n");
41 break;
42 case 1: printf("\nString(1) > String(2)\n");
43 break;
44 case -1: printf("\nString(1) < String(2)\n");
45 break;
46 }
47 }

```

运行结果:

```

C:\DS>String11
Please input string (1) : I am happy.
Please input string (2) : I am very happy.

String(1): I am happy.
String(2): I am very happy.
Compare result:
String(1) < String(2)

C:\DS>

```

### 11.5.2 抽取子字符串——Substr(char \*s1, int pos, int len)

一个字符串可能是由多个字符串所组成的,抽取子字符串则是通过截取原始字符串中一个或一个以上连续字符所组成的字符串,这些截取出来的字符串都是原始字符串的子字符串。

例如:

原始字符串:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| T | h | o | m | a | s |   | i | s |   | s  | m  | a  | r  | t  | \0 |

则其子字符串有:

|   |   |   |   |   |   |    |                   |
|---|---|---|---|---|---|----|-------------------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6  | pos: 1, length: 6 |
| T | h | o | m | a | s | \0 |                   |

|   |   |    |                   |
|---|---|----|-------------------|
| 0 | 1 | 2  | pos: 8, length: 2 |
| i | s | \0 |                   |

|   |   |   |   |   |    |                    |
|---|---|---|---|---|----|--------------------|
| 0 | 1 | 2 | 3 | 4 | 5  | pos: 11, length: 5 |
| s | m | a | r | t | \0 |                    |

在函数中输入起始位置及长度,即可得到所欲抽取之子字符串。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: String12.c */
03 /* 程序目的: 抽取子字符串 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 进行子字符串的抽取 */
08 /*-----*/
09 char *Substr(char *s,int pos,int len)
10 {
11 char s1[50]; /*声明子字符串数组*/
12 int i,j,endpos;
13
14 pos--; /*开始的位置*/
15 endpos=pos+len-1; /*结束的位置*/
16 /*将欲抽取部分复制到子字符串 substring*/
17 for (i=pos,j=0; i<=endpos; i++, j++)
18 s1[j]=s[i];
19 s1[len]='\0'; /*设置子字符串的结束字符*/
20 return s1; /*将子字符串"s1"返回给 Substr()*/
21 }
22
23 /*-----*/
24 /* 主程序:输入原始字符串, 及欲抽取的起始位置与长度, 并列出抽取结果 */
25 /*-----*/
26 void main ()
27 {
28 char string[100]; /*声明字符串数组*/
29 char *substring; /*声明字符串指针*/
30 int position; /*抽取起始位置*/
31 int length; /*抽取之子字符串长度*/
32
33 printf("\nPlease input string:");
34 gets(string); /*读取字符串存入 string*/
35 printf("Please input start position :");
36 scanf ("%d",&position); /*读取起始位置存入 position*/
37 printf("Please input substring length:");
38 scanf ("%d",&length); /*读取长度存入 length*/
39 substring = Substr(string,position,length); /*进行子字符串的抽取*/
40 printf("\nThe substring is '%s' \n",substring);

```

运行结果:

```
C:\DS>String12
Please input string: I am very happy
Please input start position : 11
Please input substring length: 5

The substring is 'happy'
C:\DS>
```

### 11.5.3 字符串的比较

字符串的比较是对原始字符串进行字符的比较, 以判断某特定字符串是否存在该原始字符串中, 若存在则此特定的字符串为原始字符串的子字符串。

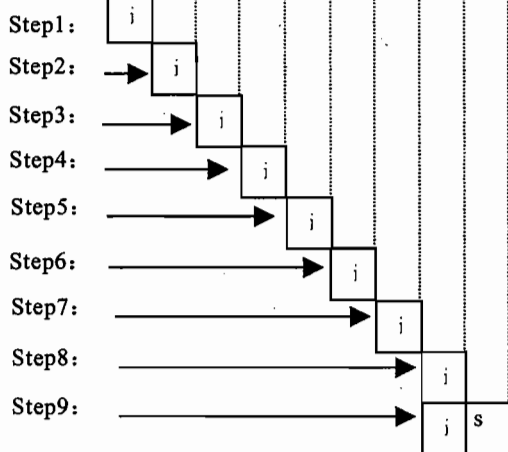
例 1:

寻找特定字符串:

|   |   |    |
|---|---|----|
| 0 | 1 | 2  |
| i | s | \0 |

原始字符串:

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| T | h | o | m | a | s |   | i | s |   | s  | m  | a  | r  | t  | \0 |

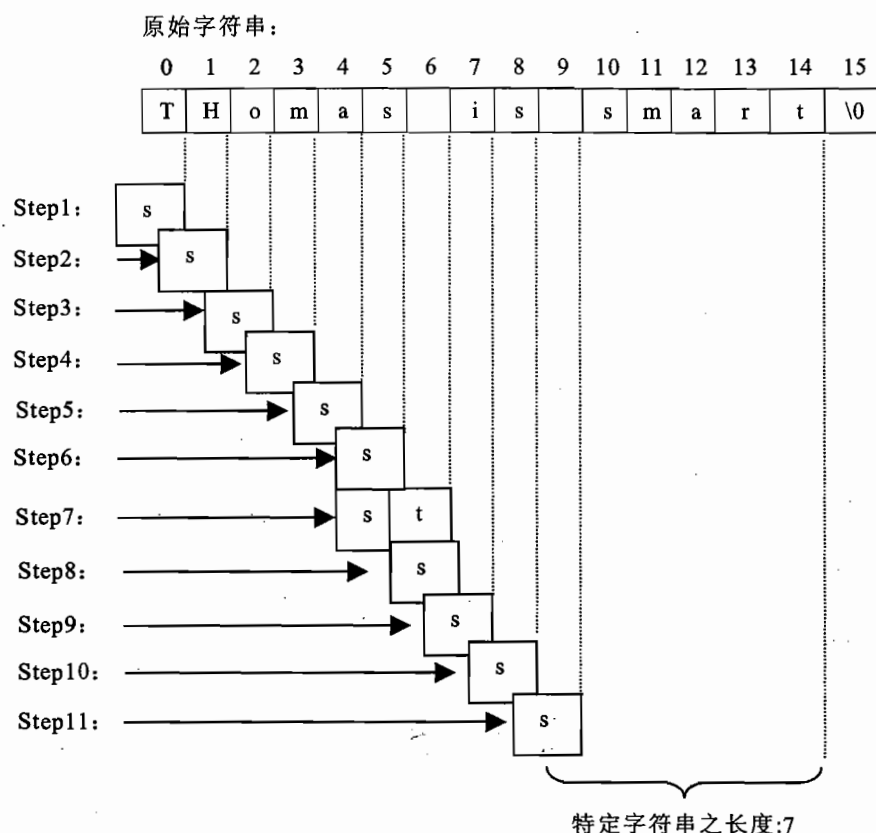


先比较特定字符串的第 1 个字符“i”, 一直比较到原始字符串的第 8 个字符才相同, 接着将特定字符串的第 2 个字符“s”和原始字符串的第 9 个字符“s”进行比较, 结果发现特定字符串存在于原始字符串中。

例 2:

寻找特定字符串:

|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6  |
| s | t | u | p | I | d | \0 |



因为特定字符串长度为 7, 故在 Step11 比较之后原始字符串的剩余比较长度已小于特定字符串的长度, 故可结束比较的动作。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: String13.c */
03 /* 程序目的: 字符串的比较 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /* 计算字符串的长度 */
08 /*-----*/
09 int Strlen(char *s)
10 {
11 int i;
12
13 /*若不为结束字符则字符串长度加1*/
14 for (i=0; s[i]!='\0';) /*用循环来计算字符串长度*/
15 i++;
16 return i; /*将累加的 i 值返回给 Strlen()*/
17 }
18
19
20 /*-----*/
21 /* 进行字符串的比较, 判断是否存在某特定字符串 */
22 /*-----*/
23 int Strstr(char *s1, char *s2)

```

```

24 {
25 int i,j,endposition;
26
27 endposition=Strlen(s1) - Strlen(s2); /*结束比较的位置*/
28
29 /*若子字符串小于字符串长度才进行比较*/
30 if (endposition > 0)
31 {
32 for (i=0 ; i<=endposition; i++)
33 {
34 /*比较各字符是否相等*/
35 for (j=i; s1[j]==s2[j-i];j++)
36 {
37 if (s2[j-i+1]=='\0') /*子字符串结束*/
38 /*返回子字符串出现的位置给 Strstr()*/
39 return i+1;
40 }
41 }
42 }
43 return -1; /*没有找到该子字符串*/
44 }
45
46 /*-----*/
47 /* 主程序:输入原始字符串.欲寻找之子字符串,并输出比较结果 */
48 /* (若该子字符串存在,则输出其在母字符串的位置) */
49 /*-----*/
50 void main ()
51 {
52 char string[100]; /*声明字符串数组*/
53 char substring[100]; /*声明子字符串数组*/
54 int final_result; /*比较的最终结果*/
55
56 /*读取字符串并存入"string"*/
57 printf("\nPlease input string:");
58 gets(string);
59
60 /*读取欲比较之子字符串*/
61 printf("\nPlease input the finding substring:");
62 gets(substring);
63
64 /*进行字符串比较*/
65 final_result=Strstr(string,substring);
66
67 if (final_result > 0)
68 /*有找到子字符串,返回其在母字符串之位置*/
69 printf("The start position of substring '%s' is [%d]\n",substring,
70 final_result);
71 else
72 /*没有找到该子字符串*/
73 printf("The substring '%s' is not in the string",substring);
74 }

```

运行结果:

```

C:\DS>String13
Please input string: This is a good book.
Please input the finding substring: good

The start position of substring 'good' is [11]

```

```
Please input string: This is a good book.
Please input the finding substring: bad

The substring 'bad' is not in the string
C:\DS>
```

### 11.5.4 字符串的分割

字符串可能是由多个子字符串所组成的，其中子字符串与子字符串间通常是用空格符分开，若欲分割字符串，则可使用空格符为分割点。在进行分割时，不管有几个连续空格符均会被忽略掉，即可得到分割的结果。

例如：

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| T | h | i | s |   | c | A | t |   |   | i  | s  |    | c  | u  | t  | e  | .  | \0 |

以空格符为分割点，得到分割的字符串如下：

|   |   |   |   |    |
|---|---|---|---|----|
| T | h | i | s | \0 |
|---|---|---|---|----|

|   |   |   |    |
|---|---|---|----|
| c | a | t | \0 |
|---|---|---|----|

|   |   |    |
|---|---|----|
| i | s | \0 |
|---|---|----|

|   |   |   |   |   |    |
|---|---|---|---|---|----|
| c | u | t | e | . | \0 |
|---|---|---|---|---|----|

程序源代码：

```
01 /*=====Program Description =====*/
02 /*程序名称: String14.c */
03 /*程序目的: 字符串的分割 */
04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* =====*/
06 /*-----*/
07 /* 进行字符串的分割 */
08 /*-----*/
09 int partition(char *s1,char *s2,int pos)
10 {
11 int i,j;
12
13 i=pos; /*从欲分割的位置开始进行分割*/
14 while (s1[i] == ' ') /*忽略字符串前的所有空格符*/
15 i++;
16 if (s1[i] != '\0') /*判断字符串是否已结束*/
17 {
18 j=0;
19 /*复制非空格符直到找到下一个空格符*/
20 while (s1[i] != '\0' && s1[i] != ' ')
21 {
22 s2[j] = s1[i];
23 i++;
24 j++;
25 }
26 s2[j]='\0'; /*设置分割字符串之结束字符*/
```

```

27 return I; /*返回目前的位置给 partition()*/
28 }
29 else
30 return -1; /*结束分割*/
31 }
32 /*-----*/
33 /* 主程序:输入原始字符串,进行分割后输出分割结果 */
34 /*-----*/
35 void main ()
36 {
37 char string[50]; /*声明字符串数组*/
38 char partition_string[20]; /*声明分割字符串数组*/
39 int position; /*分割的位置*/
40 int k;
41
42 printf("\nPlease input string :");
43 gets(string); /*读取字符串存入 string*/
44 position=0; /*设置进行分割的第一个位置*/
45 printf("\nPartition result:\n");
46
47 /*进行字符串分割,直到字符串结束*/
48 k=0;
49 while ((position = partition(string,partition_string,
50 position)) != -1)
51 { k++;
52 /*输出各分割出来的子字符串*/
53 printf("Partition %d :%s\n",k,partition_string);
54 }
55 }

```

运行结果:

```

C:\DS>String14
Please input string :Thomas is a smart boy

Partition result:
Partition 1 : Thomas
Partition 2 : is
Partition 3 : a
Partition 4 : smart
Partition 5 : boy

C:\DS>

```

### 11.5.5 常用的字符串函数

综合常用的函数,整理如下表:

|   | 函数名称   | 功能说明                |
|---|--------|---------------------|
| 1 | Strlen | 计算字符串长度,不包含结束字符“\0” |
| 2 | Strcat | 将两个字符串合并成一个字符串      |
| 3 | Strins | 插入字符串               |
| 4 | Strdel | 删除字符串中的子字符串         |



续表

|    | 函数名称   | 功能说明                      |
|----|--------|---------------------------|
| 5  | Strcpy | 将一个字符串复制到另一个字符串           |
| 6  | Strcmp | 比较两个字符串, 大小写视为不同          |
| 7  | Strrep | 将字符串中的某一子字符串用另一个子字符串的内容替换 |
| 8  | Substr | 抽取出子字符串                   |
| 9  | Strstr | 找出子字符串在母字符串中第一次出现的位置      |
| 10 | Strlwr | 将字符串内所有字符转小写              |
| 11 | Strupr | 将字符串内所有字符转大写              |
| 12 | Strset | 将字符串所有字符设为指定的字符           |
| 13 | Strrev | 将字符串反转                    |

其中 1~9 项在前面章节已对字符串处理方法做详细介绍, 10~13 项就留给读者自行练习。

这些函数的原型都是定义在<string.h>头文件中, 所以使用到这些函数时, 程序必须引含<string.h>。

## 11.6 字符串转换数值的应用

由数字所组成的字符串, 例如“96”、“123.56”等常会为了计算必须转换成数值, 将字符串“96”转换成整数的 96, 将字符串“123.56”转换成浮点数的 123.56, 才能对其进行数值运算。本节将介绍几个字符串转换数值的函数, 提供参考。

函数 1:

```
#include <math.h>
double atof(const char *s)
```

将 s 字符串转换并返回双精度浮点数。转换不成功时则返回零(0)值

函数 2:

```
#include <stdlib.h>
int atoi(const char *s)
```

将 s 字符串转换并返回整数值。转换不成功时则返回零(0)值

函数 3:

```
#include <stdlib.h>
long atol(const char *s)
```

将 s 字符串转换并返回长整数值。转换不成功时则返回零(0)值

以下的程序将已知的字符串使用 atoi()、atol()、atof()函数转换为数值。

程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: String15.c */
03 /* 程序目的: 字符串的数值转换 */
```

```

04 /* Written By Nai-Ging Yeh. (Cherish Studio.) */
05 /* ===== */
06 /*-----*/
07 /*将已知字符串使用 atoi ()、atol ()、atof ()函数转换为数值 */
08 /*-----*/
09 #include <stdlib.h>
10 main ()
11 {
12 char s1[3]="50", s2[6]="54321", s3[7]="987.32";
13 int a;
14 long int b;
15 double c;
16 a=atoi (s1); /*整数*/
17 b=atol (s2); /*长整数*/
18 c=atof (s3); /*双精度浮点数*/
19 printf("String '%s' is value [%d]\n",s1,a);
20 printf("String '%s' is value [%d]\n",s2,b);
21 printf("String '%s' is value [%d]\n",s3,c);
22 }

```

运行结果:

```

C:\DS>String15
String '50' is value [50]
String '54321' is value [54321]
String '987.32' is value [987.32]

C:\DS>

```

## 【习题】

一、复习:

1. C 语言的字符串结束符号为\_\_\_\_\_。
2. 字符串在函数间的传递方式是以\_\_\_\_\_的方式传递。
3. 字符串函数大都定义于头文件\_\_\_\_\_中。
4. 字符串的输入函数有两种: \_\_\_\_\_、\_\_\_\_\_。
5. 字符串的输出函数有两种: \_\_\_\_\_、\_\_\_\_\_。
6. 字符串是由一连串的数值所组成的集合。
7. 字符串的表示方式有“数组结构”和“指针”两种。
8. strlen( )函数是用来计算字符串长度,其中包含结束字符“\0”。
9. 下列那些是正确的字符串类型声明?
  - (a) char string[ ];
  - (b) char \*string[10];
  - (c) char \*string;
  - (d) char string[10];
  - (e) char \*string[ ];
10. 下列那些是正确的字符串输入指令?
 

```
char *s, x[50];
```

  - (a) scanf(s);
  - (b) scanf("%s",s);

- (c) gets(x);  
 (d) gets("%s",x);  
 (e) scanf("%s",\*s);
11. 下列那些指定语句是正确的指定?  
 char \*p = "happy";  
 char q[10] = "lucky";  
 char \*r;  
 char s[20];  
 (a) r=q    (b) r=p    (c) s=q    (d) s=p
12. 请问下列字符串在主存储器中占多少字节的空间? 字符串长度各为多少?  
 (a) char string[10] = "happy";  
 (b) char string[80] = "happy\_girl";  
 (c) char string[ ] = "smart";  
 (d) char \*string = "computer";  
 (e) char string[5] = " ";

## 二、应用:

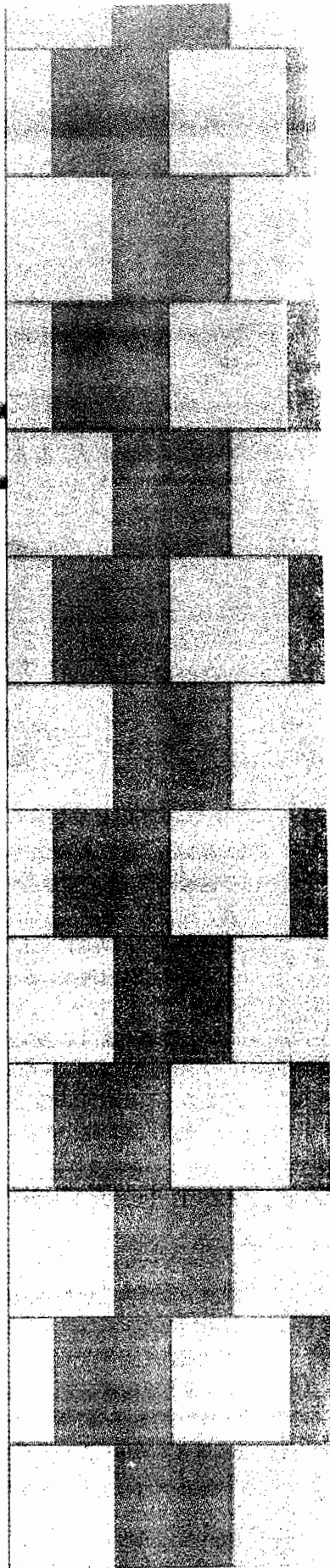
1. 请写出下列程序的运行结果:  

```
#include <string.h>
main ()
{
 char string1[] = "c language ";
 char string2[] = "data structure";
 strcat(string1,string2);
 printf("%s %d \n",string1,strlen(string1));
}
```
2. 请分别写出 C 语言中的字符串函数之程序。  
 (1) strlen()  
 (2) strcat()  
 (3) strcmp()
3. 请设计出一函数将字符串的内容反转。
4. 接上题, 试写一个程序:  
 输入一个 5 位数字, 将该数字反转, 与原数相加后输出。  
 例如:
- |       |           |
|-------|-----------|
| 输入原数: | 3 6 2 9 4 |
| 反转:   | 4 9 2 6 3 |
| 相加:   | 8 5 5 5 7 |
5. 请设计一函数以用来删除字符串内之空格符。  
 (1) 删除一个已知字符串前端的空格符并返回。  
 (2) 删除一个已知字符串尾端的空格符并返回。  
 (3) 综合上述两项功能。

# 图 形 结 构

## 第 12 章

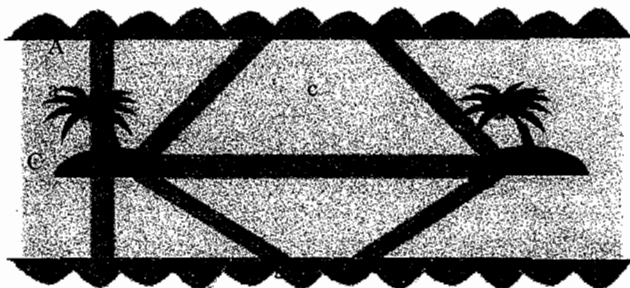
- ◆ 何谓图形结构
- ◆ 图形的表示法
- ◆ 图形的查找
- ◆ 生成树问题
- ◆ 最短路径问题



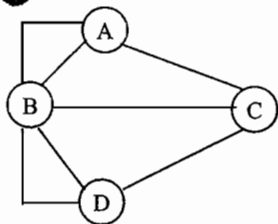
## 12.1 何谓图形结构

图形问题的讨论,最早可追溯到公元 1736 年,数学家欧拉(Euler)为了解“7 桥问题”(Koenigsberg Seven Bridge Problem)所设计出了解法。在开始图形结构这一章之前,我们先来看一看欧拉当时所解的 7 桥问题。

古时候在 Koenigsberg 这个地方,有一条河,河中有一座岛,除了河所区别的两岸和小岛之外,还有一个半岛紧邻着河流,因此区别出 4 个区域(A、B、C、D),连接这 4 个区域的是 7 座桥(a、b、c、d、e、f、g),如下图所示。而 7 桥问题是“是否可以从小某一个区域出发,经过每座桥一次,而回到原先出发的区域?”



后来欧拉将 7 桥问题所区别的 4 个区域,定义成图形中的 4 个顶点(Vertexes),把 7 座桥定义成图形中的 7 个边(Edges),最后他发现如果任一区域(顶点)上所连结的桥数(边数)为偶数,才有可能从任一桥(顶点)出发,经过每一座桥(边)而回到出发的区域(顶点),我们称这种走法为“欧拉走法(Eulerian Walk)”。欧拉的 7 桥问题图形如右:



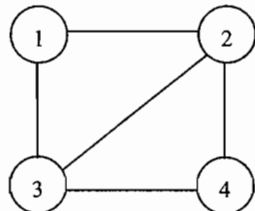
由欧拉的发现中,我们可以了解到 7 桥问题是没有解的,因为在 7 桥问题中,每一个顶点皆是奇数个边。

从 7 桥问题中,我们已经对图形有了基本的认识了。图形的定义为“在图形 G 中包含了两个集合,一个是由顶点(Vertexes 或 nodes)所构成的有限的非空集合,另一个是由边(Edges 或 Arcs)所构成的有限非空集合。”我们可以  $G(V,E)$  来表示。

在进入图形这个主题之前,我们必须先定义出在图形中常用的名词,以便在之后的章节运用。

### 12.1.1 无向图形

所谓的无向图形(Undirected Graph)是指在图形中任一顶点上的边都是没有方向性的。例如右图就是一个无向图形,因为任意两个顶点间的边都没有方向性,如顶点 1 和顶点 2 之间的边表示可从顶点 1 到顶点 2,也可以从顶点 2 到顶点 1。



从集合的概念中,我们可以将上图表示为:

$$V(G) = \{1, 2, 3, 4\}$$

$$E(G) = \{ (1, 2), (1, 3), (2, 3), (2, 4), (3, 4) \}$$

## 12.1.2 有向图形

所谓的有向图形(Directed Graph)是指在图形中任一顶点上的边都是有方向性的。例如右图就是一个有向图形, 因为任意两个顶点间的边都是有方向性, 如顶点 1 和顶点 2 之间的边表示可从顶点 1 到顶点 2, 但是不可以从顶点 2 到顶点 1。

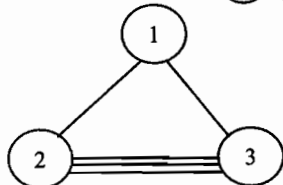
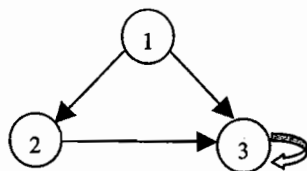
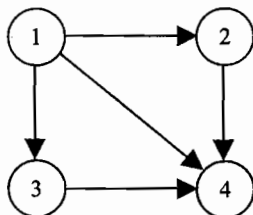
根据集合的概念, 我们可以将上图表示为:

$$V(G) = \{1, 2, 3, 4\}$$

$$E(G) = \{ \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 4 \rangle, \langle 3, 4 \rangle \}$$

注意: 在无向图形和有向图形之中, 我们还必须加上一些限制:

1. 图形中不允许自身循环(Self Loop)如右图的顶点 3:
2. 图形中两顶点间的边, 不可重复。边重复的图形, 我们称为多边形(Multi-Graph), 如右图顶点 2 到顶点 3, 有 3 条重复的边。



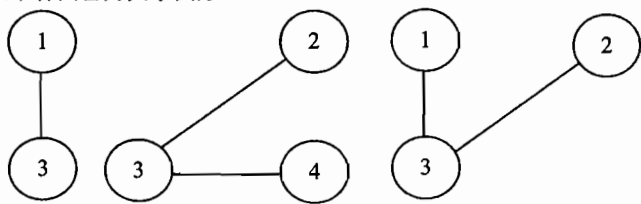
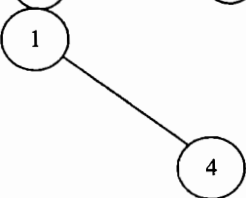
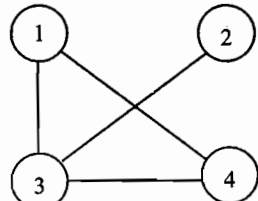
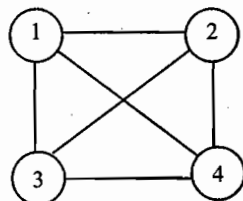
## 12.1.3 完全图形

所谓的完全图形(Complete Graph)是指在无向图形中任意两顶点上的都存在一个边。例如右图就是一个完全图形, 因为任意两个顶点间的都存在一个边。

## 12.1.4 子图形

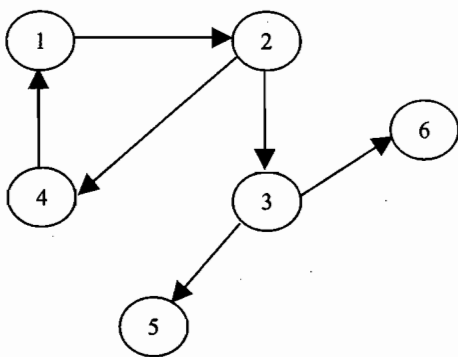
所谓的子图形(Sub-Graph)是指由图形 G 中取出的部分集合, 如右图为图形 G。

则以下各图皆为其子图形。



## 12.1.5 路径

所谓的路径是指在图形中从顶点 A 到达顶点 B, 所经过上的所有的边。例如下图从顶点 1 到顶点 5 的路径为  $\langle V_1, V_2 \rangle, \langle V_2, V_3 \rangle, \langle V_3, V_5 \rangle$ , 而路径的长度为经过的边数, 在这个例子中, 路径的长度为 3。

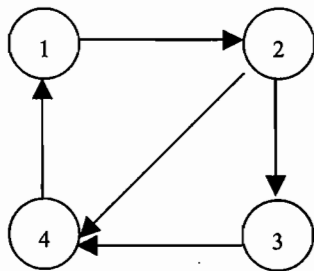


### 12.1.6 简单路径

所谓的简单路径(Simple Path)是指在图形中，除了起点和终点可以重复(不重复亦可)外，其余的顶点皆不相同的路径，如上图的  $\langle V1, V2 \rangle, \langle V2, V3 \rangle, \langle V3, V5 \rangle$ ，为简单路径，而  $\langle V1, V2 \rangle, \langle V2, V4 \rangle, \langle V4, V1 \rangle, \langle V1, V2 \rangle, \langle V2, V3 \rangle, \langle V3, V5 \rangle$  不是简单路径，因为在这条路径中顶点 1 和顶点 2 重复经过。

### 12.1.7 回路

所谓的回路(Cycle)是指在图形中，起点和终点相同的简单路径，如右图中， $\langle V1, V2 \rangle, \langle V2, V4 \rangle, \langle V4, V1 \rangle$  就是一条回路、而  $\langle V1, V2 \rangle, \langle V2, V3 \rangle, \langle V3, V4 \rangle, \langle V4, V1 \rangle$ ，也是一条回路。



### 12.1.8 连通顶点

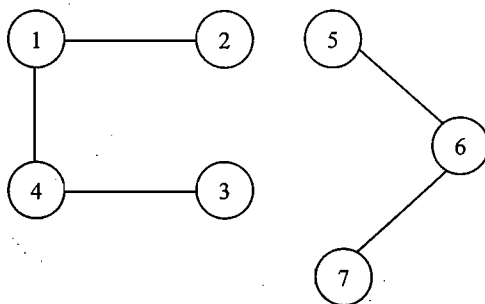
所谓的连通顶点(Connected Vertices)是指在无向图形中，顶点 A 到顶点 B 间存在一条路径，则称顶点 A 和顶点 B 为连通顶点。

### 12.1.9 连通图形

如果在无向图形中，任意两个顶点间皆连通，则称为连通图形(Connected Graph)。即任意两个顶点皆存在有一条路径可到达。

### 12.1.10 连通单元

所谓的连通单元(Connected Component)是将无向图形分为多个分离的子图形之后，原图形的连通顶点仍在同一个子图中，如下图所示。



### 12.1.11 强连通顶点

所谓的强连通顶点(Strongly Connected Vertices)是指在有向图形中, 顶点 A 到顶点 B 间存在一条路径, 而顶点 B 到顶点 A 间也存在一条路径, 则称顶点 A 和顶点 B 为强连通顶点。

### 12.1.12 强连通图形

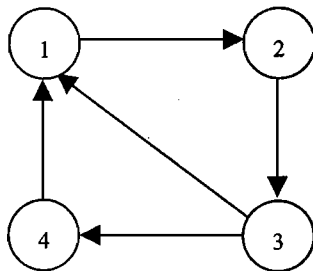
如果在有向图形中, 任意两个顶点间皆存在一条路径可到对方, 则称为强连通图形(Strongly Connected Graph)。

### 12.1.13 强连通单元

所谓的强连通单元(Connected Component)是将有向图形, 分为多个分离的子图形之后, 原图形的连通顶点仍在同一个子图中。

除了以上的定义外, 对于特殊的图形结构。如树状结构的图形, 我们定义为: 自由树是指一个相连非循环的无向图形。如果一棵自由树, 有一个特定的顶点作为树根, 则称为有根树(Rooted Tree)。

在有向图形中, 因为边皆是有方向性的, 为了需要我们还定义出了内分支度(In-degree)和外分支度(Out-degree)。内分支度是指由其它顶点前往此顶点的边数、而外分支度则是指由此顶点前往其它顶点的边数。如右图: 顶点 1 的内分支度为 2, 外分支度为 1。在有向图形中, 各顶点的内分支度或外分支度总和, 即为此图形的边数。

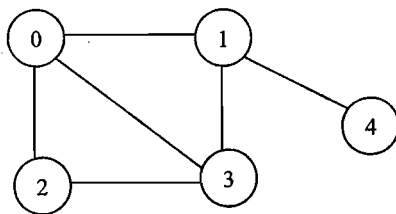


## 12.2 图形的表示法

### 12.2.1 邻接数组表示法

邻接数组表示法(Adjacent Matrix)是以一个  $n \times n$  的数组来表示一个具有  $n$  个顶点的图形。我们以数组的索引值来表示顶点、以数组的内容值来表示顶点间的边是否存在(以 1 表示存在边, 以 0 表示不存在边)。如右图的无向图形:

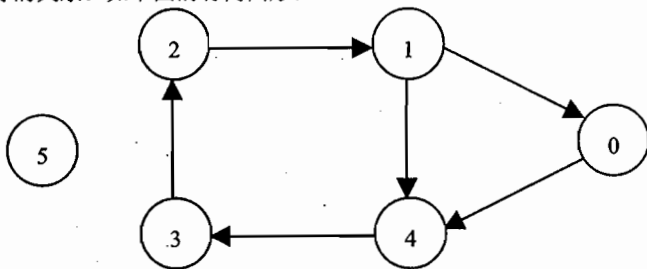
其邻接数组为:





|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 2 | 1 | 0 | 0 | 1 | 0 |
| 3 | 1 | 1 | 1 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 |

在无向图形中，邻接数组中的内容值所呈现出的是一个对称的关系，因为如果顶点 A 和顶点 B 存在公共边，表示顶点 A 可到达顶点 B、顶点 B 也可到达顶点 A。但是在有向图形，邻接数组中的内容值并不一定会呈现出对称的关系。如下图的有向图形：



其邻接数组为：

|   | 0 | 1        | 2        | 3        | 4 | 5 |
|---|---|----------|----------|----------|---|---|
| 0 | 0 | 0        | 0        | 0        | 1 | 0 |
| 1 | 1 | 0        | 0        | 0        | 1 | 0 |
| 2 | 0 | <u>1</u> | 0        | 0        | 0 | 0 |
| 3 | 0 | 0        | <u>1</u> | 0        | 0 | 0 |
| 4 | 0 | 0        | 0        | <u>1</u> | 0 | 0 |
| 5 | 0 | 0        | 0        | 0        | 0 | 0 |

程序实例：

设计一个将上述图形转成邻接数组的程序。

程序构思：

用户输入与各个边，再将边转成邻接数组：

程序源代码：

```

01 /* ===== Program Description ===== */
02 /* 程序名称: m_Graph.c */
03 /* 程序目的: 设计一个将图形转成邻接数组的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #define Max 6 /* 定义最大可输入数 */
07
08 int Graph[Max][Max]; /* 图形邻接数组 */
09 /* ----- */
10 /* 输出邻接数组数据 */
11 /* ----- */

```

```

12 void Print_M_Graph()
13 {
14 int i,j;
15
16 printf("Vertice");
17 for (i=0;i<Max;i++)
18 printf("%3d",i);
19 printf("\n");
20 for (i=0;i<Max;i++)
21 {
22 printf("%4d ",i);
23 for (j=0;j<Max;j++)
24 printf("%3d",Graph[i][j]);
25 printf("\n");
26 }
27 }
28
29 /* ----- */
30 /* 以邻接数组建立图形 */
31 /* ----- */
32 void Create_M_Graph(int Vertice1,int Vertice2)
33 {
34
35 Graph[Vertice1][Vertice2] = 1; /* 将数组内容设为1 */
36 }
37
38 /* ----- */
39 /* 主程序 */
40 /* ----- */
41 void main ()
42 {
43 int Source; /* 起始顶点 */
44 int Destination; /* 终止顶点 */
45 int i,j;
46
47 for (i=0;i<Max;i++)
48 for (j=0;j<Max;j++)
49 Graph[i][j] = 0;
50 while (1)
51 {
52 printf("Please input the Edge's source : ");
53 scanf("%d",&Source);
54 if (Source == -1)
55 break;
56
57 printf("Please input the Edge's Destination : ");
58 scanf("%d",&Destination);
59
60 if (Source == Destination) /* 错误: 自身循环 */
61 printf("@Error@ : Self Loop!!\n");
62 /* 错误: 超出范围 */
63 else if (Source >= Max || Destination >= Max)
64 printf("@Error@ : Out of range!!\n");
65 else /* 调用建立邻接数组 */
66 Create_M_Graph(Source, Destination);
67 }
68 printf("##Graph##\n");
69 Print_M_Graph(); /* 调用输出邻接数组数据 */
70 }

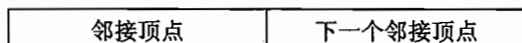
```

运行结果:

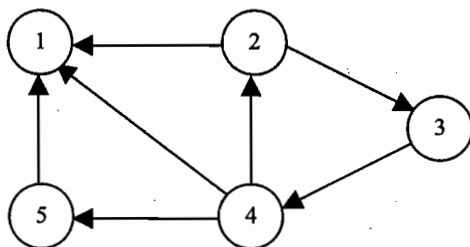
```
C:\DS>m_graph
Please input the Edge's source : 0
Please input the Edge's Destination : 4
Please input the Edge's source : 1
Please input the Edge's Destination : 0
Please input the Edge's source : 1
Please input the Edge's Destination : 4
Please input the Edge's source : 2
Please input the Edge's Destination : 1
Please input the Edge's source : 3
Please input the Edge's Destination : 2
Please input the Edge's source : 4
Please input the Edge's Destination : 3
Please input the Edge's source : -1
##Graph##
Vertex 0 1 2 3 4 5
0 0 0 0 0 1 0
1 1 0 0 0 1 0
2 0 1 0 0 0 0
3 0 0 1 0 0 0
4 0 0 0 1 0 0
5 0 0 0 0 0 0
C:\DS>
```

## 12.2.2 邻接列表表示法

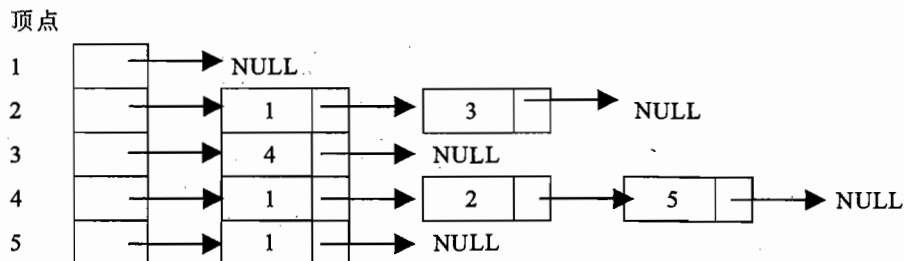
邻接列表法(Adjacency List)是以链表来记录各顶点的邻接顶点。其节点结构如下:



如下图的有向图形:



其邻接列表为:



在C语言中,邻接列表的结构声明如下:

```
struct Node
{
 int Vertex;
 struct Node *Next;
};
typedef struct Node *Graph;
struct Node Head[VertexNum];
```

#### 程序实例:

设计一个将上述图形转成邻接列表的程序。

#### 程序构思:

用户输入与各个边,再将边转成邻接列表。

邻接列表结构的声明:

```
struct Node
{
 int Vertex;
 struct Node *Next;
};
typedef struct Node *Graph;
struct Node Head[VertexNum];
```

邻接列表的建立:

将新输入的节点(邻接顶点)插入在原顶点列表尾端(请参考链表的插入)。

#### 程序源代码:

```
01 /* ===== Program Description ===== */
02 /* 程序名称: l_Graph.c */
03 /* 程序目的: 设计一个将图形转成邻接列表的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define VertexNum 6 /* 定义顶点数 */
08
09 struct Node /* 声明图形顶点结构 */
10 {
11 int Vertex; /* 邻接顶点数据 */
12 struct Node *Next; /* 下一个邻接顶点 */
13 };
14 typedef struct Node *Graph; /* 定义图形结构 */
15 struct Node Head[VertexNum]; /* 顶点数组 */
16
17 /* ----- */
18 /* 建立邻接顶点至邻接列表内 - */
19 /* ----- */
20 void Create_L_Graph(int Vertex1,int Vertex2)
21 {
22 Graph Pointer; /* 节点声明 */
23 Graph New; /* 新顶点声明 */
24
25 New = (Graph) malloc(sizeof(struct Node)); /* 配置内存 */
26 if (New != NULL) /* 配置成功 */
27 {
```

```

28 New->Vertex = Vertex2; /* 邻近顶点 */
29 New->Next = NULL; /* 下一个邻接顶点指针
30 /**/ Pointer 指针设为顶点数组之首节点 */
31 Pointer = &(Head[Vertex1]);
32
33 while (Pointer->Next != NULL)
34 Pointer = Pointer->Next; /* 往下一个节点 */
35
36 Pointer->Next = New; /* 串连在链接尾端 */
37 }
38 }
39
40 /* ----- */
41 /* 输出邻接列表内数据 */
42 /* ----- */
43 void Print_L_Graph(struct Node *Head)
44 {
45 Graph Pointer; /* 节点声明 */
46
47 Pointer = Head->Next; /* Pointer 指针设为首节点 */
48 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
49 {
50 printf("[%d]", Pointer->Vertex);
51 Pointer = Pointer->Next; /* 往下一个节点 */
52 }
53 printf("\n");
54 }
55
56 /* ----- */
57 /* 主程序 */
58 /* ----- */
59 void main ()
60 {
61 int Source; /* 起始顶点 */
62 int Destination; /* 终止顶点 */
63 int i, j;
64
65 for (i=0; i<VertexNum; i++)
66 {
67 Head[i].Vertex = i;
68 Head[i].Next = NULL;
69 }
70 while (1)
71 {
72 printf("Please input the Edge's source : ");
73 scanf("%d", &Source);
74 if (Source == -1)
75 break;
76
77 printf("Please input the Edge's Destination : ");
78 scanf("%d", &Destination);
79
80 /* 错误: 超出范围 */
81 if (Source >= VertexNum || Destination >= VertexNum)
82 printf("@Error@ : Out of range!!\n");
83 else /* 调用建立邻接列表 */
84 Create_L_Graph(Source, Destination);
85 }
86 printf("###Graph###\n");
87 for (i=0; i<VertexNum; i++)
88 {

```

```

89 printf("Vertex[%d] : ",i);
90 Print_L_Graph(&Head[i]);/* 调用输出邻接列表数据 */
91 }
92 }

```

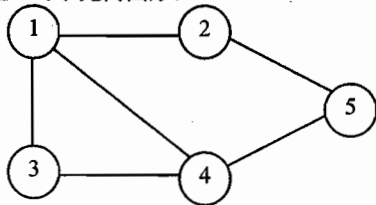
运行结果:

```

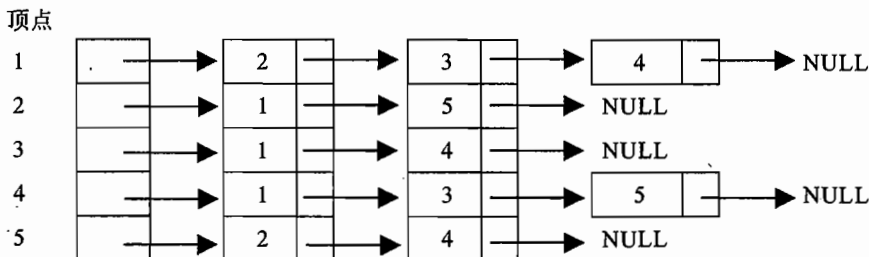
C:\DS>l_graph
Please input the Edge's source : 2
Please input the Edge's Destination : 1
Please input the Edge's source : 2
Please input the Edge's Destination : 3
Please input the Edge's source : 3
Please input the Edge's Destination : 4
Please input the Edge's source : 4
Please input the Edge's Destination : 1
Please input the Edge's source : 4
Please input the Edge's Destination : 2
Please input the Edge's source : 4
Please input the Edge's Destination : 5
Please input the Edge's source : 5
Please input the Edge's Destination : 1
Please input the Edge's source : -1
##Graph##
Vertex[0] :
Vertex[1] :
Vertex[2] : [1][3]
Vertex[3] : [4]
Vertex[4] : [1][2][5]
Vertex[5] : [1]
C:\DS>

```

如果邻接列表所存储的是一个无向图形,因为无向图形中的边都是对称的,所以用邻接列表时,除了邻接顶点外还存储一个指针。我们也可以将无向图的邻接列表用邻接列表循序表示法“(Sequential representation of Adjacency Lists)”来表示。所谓的邻接列表循序表示法,是将邻接列表中的内容存储一维数组中,如果是一个有  $n$  个顶点和  $e$  个边的无向图,我们需要一个大小为  $n+2e+1$  的一维数组,前  $n+1$  个数组元素存储列表的开头位置。对于无向图形:



其邻接列表为:



其邻接列表循序表示法为:

步骤 1: 声明一个  $n+2e+1$  的数组。

$$n+2e+1=5+2*6+1=18$$

步骤 2: 前  $n+1$  个元素存储列表的开头位置。

先空下  $\text{Data}[0]$  到  $\text{Data}[5]$  元素。

步骤 3: 将列表数据依序存储于数组第  $n+1$  个元素之后。

列表数据序为 2、3、4、1、5、1、4、1、3、5、2、4。

步骤 4: 存储列表的开头位置。

第一个顶点的起始位置为 6, 数组第一个元素存 6。

第二个顶点的起始位置为 9, 数组第二个元素存 9。

第 3 个顶点的起始位置为 11, 数组第 3 个元素存 11。

第 4 个顶点的起始位置为 13, 数组第 4 个元素存 13。

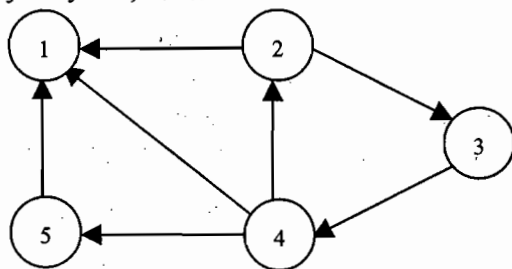
第 5 个顶点的起始位置为 16, 数组第 5 个元素存 16。

步骤 5: 第  $n+1$  元素存储  $n+2e+1$ 。

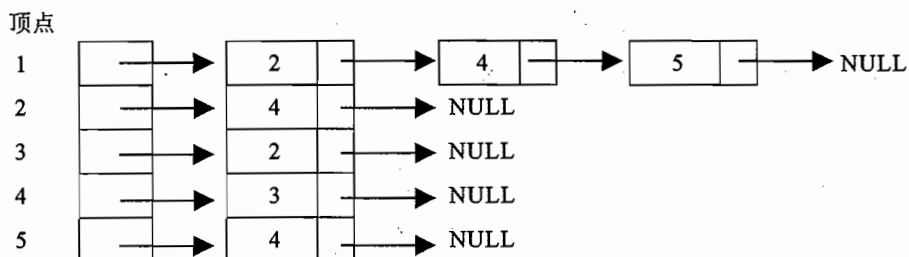
第 6 个元素存储 18。

| Data | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] | [15] | [16] | [17] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|
|      | 6   | 9   | 11  | 13  | 16  | 18  | 2   | 3   | 4   | 1   | 5    | 1    | 4    | 1    | 3    | 5    | 2    | 4    |

我们从遍历邻接列表的过程中得知一个顶点的节点数, 就可以算出其外分支度, 但是却无法算出其内分支度。为了内分支度的计算, 我们还可利用另一个列表来记录到达该顶点的顶点数据, 我们称这种列表为反转邻接列表(Inverse Adjacency Lists), 如有向图形:



其反转邻接列表为:

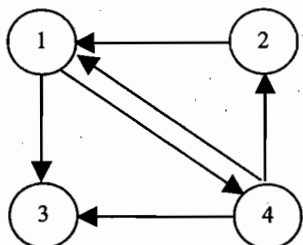


除了以上的邻接列表表示法之外, 我们可以运用另外的节点结构来存储顶点间的关系。我们可以定义另一种节点结构为:

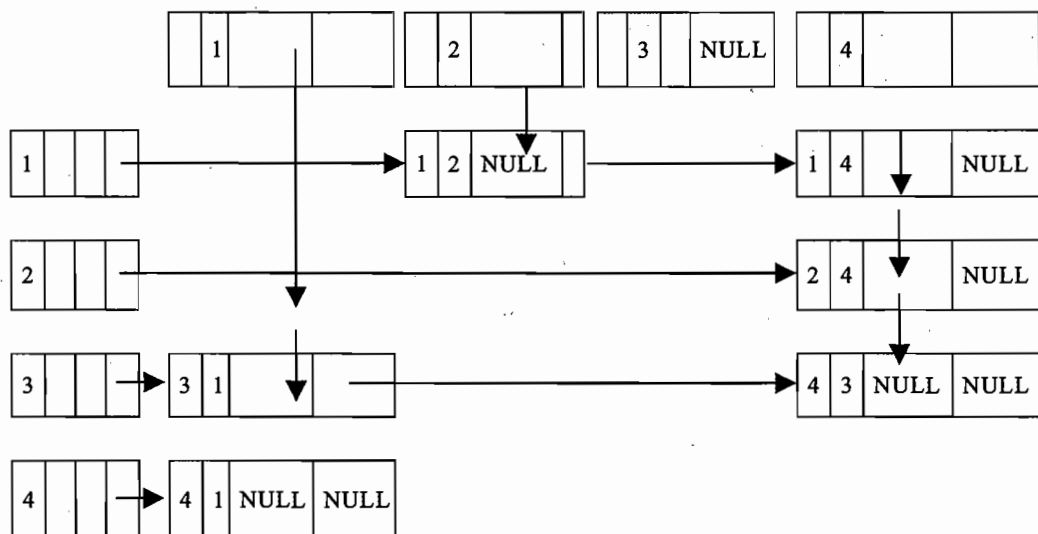
| 起始顶点 | 终止顶点 | 栏指针 | 列指针 |
|------|------|-----|-----|
|------|------|-----|-----|

我们称这种表示法为正交列表表示法(Orthogonal List representation)。

如有向图形:



其正交列表表示法为:

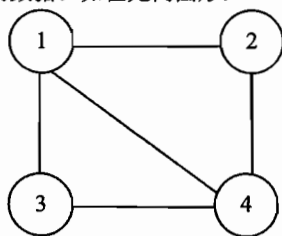


### 12.2.3 多重邻接列表表示法

在无向图形的邻接列表表示法当中, 每一个边都会出现两次。有些时候为了处理的方便, 我们必须快速的找到一个边的第二项, 并加上标记表示已处理过。此时我们可以运用多重邻接列表来记录图形, 在多重邻接列表中节点可以在多个列表中使用。多重邻接列表的节点结构如下:

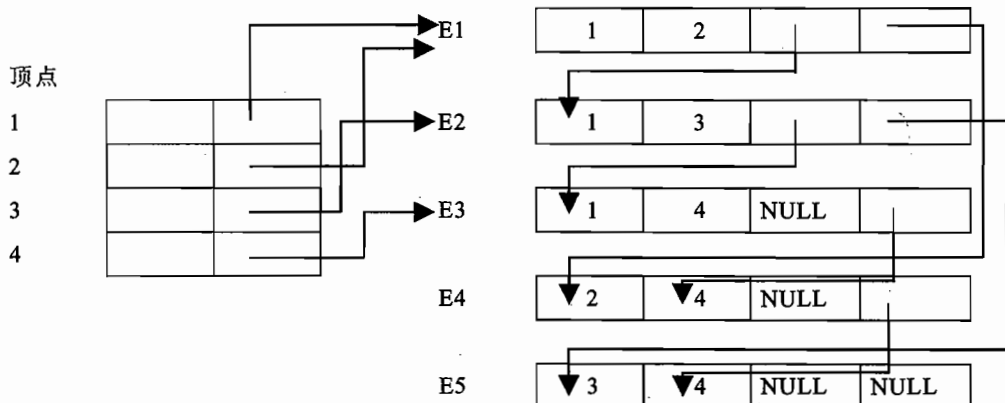
| Marked | Vertex1 | Vertex2 | Link1 | Link2 |
|--------|---------|---------|-------|-------|
|--------|---------|---------|-------|-------|

每一个节点记录着一个边的数据。如: 在无向图形:





有 5 条边  $E1=(1,2)$ 、 $E2=(1,3)$ 、 $E3=(1,4)$ 、 $E4=(2,4)$ 、 $E5=(3,4)$ 。其多重邻接列表为：



我们从多重邻接列表中，我们可以轻易的发现：

与顶点 1 相邻的边有：E1、E2、E3。

与顶点 2 相邻的边有：E1、E4。

与顶点 3 相邻的边有：E2、E5。

与顶点 4 相邻的边有：E3、E4、E5。

在 C 语言中，多元邻接列表的结构声明如下：

```
struct Edge
{
 int Marked;
 int Vertex1;
 int Vertex2;
 struct Edge *Edge1;
 struct Edge *Edge2;
};
typedef struct Edge *NextEdge;

struct Node
{
 int Vertex;
 struct Edge *Edge;
}
typedef struct Node *Graph;
struct Node Head[VerticeNum];
```

程序实例：

设计一个将上述图形转成多元邻接列表的程序。

程序构思：

用户输入与各个边，再将边转成多元邻接列表。

多元邻接列表结构的声明：

```
struct Edge
{
 int Marked;
 int Vertex1;
```

```

 int Vertex2;
 struct Edge *Edge1;
 struct Edge *Edge2;
 };
 typedef struct Edge *NextEdge;

 struct Node
 {
 int Vertex;
 struct Edge *Edge;
 }
 typedef struct Node *Graph;
 struct Node Head[VertexNum];

```

多元邻接列表的建立:

声明一个新的多元邻接列表节点(New),  
 将 New->Vertex1 设为用户输入的起始顶点。  
 将 New->Vertex2 设为用户输入的终止顶点。  
 将 New->Edge1 设为 NULL。  
 将 New->Edge2 设为 NULL。

无向图形:

如果列表未有节点,则将新节点串连至起始顶点的 Head 之后。

如果列表有节点,且列表尾端的 Vertex1 为起始顶点,则往 Edge1 的所指的节点(下一个节点),否则往 Edge1 的所指的节点(下一个节点)。重复此步骤直到列表尾端为止。

如果列表尾端的 Vertex1 为起始顶点,则将列表尾端的 Edge1 指针指向新节点。否则将列表尾端的 Edge2 指针指向新节点。

在无向图形中,因为起始顶点和终止顶点皆可通连,所以再做一次终止顶点的查找,重复以上 3 个步骤。

如果有向图形,无法节省任何节点。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: ml_Graph.c */
03 /* 程序目的: 设计一个将图形转成多元邻接列表的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define VertexNum 6 /* 定义顶点数 */
08
09 struct Edge
10 {
11 int Marked;
12 int Vertex1;
13 int Vertex2;
14 struct Edge *Edge1;
15 struct Edge *Edge2;
16 };
17 typedef struct Edge *NextEdge;
18
19 struct Node /* 声明图形顶点结构 */
20 {
21 int Vertex; /* 邻接顶点数据 */

```

```

22 struct Edge *Next; /* 下一个邻接顶点 */
23 };
24 typedef struct Node *Graph; /* 定义图形结构 */
25 struct Node Head[VertexNum]; /* 顶点数组 */
26
27 /* ----- */
28 /* 建立邻接点至邻接列表内 */
29 /* ----- */
30 void Create_ML_Graph(int Vertex1, NextEdge New)
31 {
32 NextEdge Pointer; /* 节点声明 */
33 NextEdge Previous; /* 前一个节点 */
34
35 Previous = NULL;
36
37 Pointer = Head[Vertex1].Next; /* 目前的节点 */
38
39 while (Pointer != NULL) /* 到达列表尾端才结束循环 */
40 {
41 Previous = Pointer;
42 /* 如果 Vertex1 为起始顶点 */
43 if (Pointer->Vertex1 == Vertex1)
44 /* 往 Edge1 的下一个节点 */
45 Pointer = Pointer->Edge1;
46 else
47 /* 往 Edge2 的下一个节点 */
48 Pointer = Pointer->Edge2;
49 }
50 if (Previous == NULL) /* 串连在 Head 之后 */
51 Head[Vertex1].Next = New;
52 else if (Previous->Vertex1 == Vertex1)
53 Previous->Edge1 = New; /* 串连在 Edge1 之后 */
54 Else
55 Previous->Edge2 = New; /* 串连在 Edge2 之后 */
56 }
57
58 /* ----- */
59 /* 输出邻接列表内数据 */
60 /* ----- */
61 void Print_ML_Graph(struct Node *Head)
62 {
63 NextEdge Pointer; /* 节点声明 */
64
65 Pointer = Head[0].Next; /* Pointer 指针设为首节点 */
66 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
67 {
68 printf("%d,%d", Pointer->Vertex1, Pointer->Vertex2);
69 if (Head[0].>Vertex == Pointer->Vertex1)
70 Pointer = Pointer->Edge1; /* 往下一个节点 */
71 else if (Head[0].>Vertex == Pointer->Vertex2)
72 Pointer = Pointer->Edge2;
73 }
74 printf("\n");
75 }
76
77 /* ----- */
78 /* 主程序 */
79 /* ----- */
80 void main ()
81 {
82 int Source; /* 起始顶点 */

```

```

83 int Destination; /* 终止顶点 */
84 int Choose; /* 选项变量 */
85 NextEdge New; /* 新节点 */
86 int i,j;
87
88 for (i=0;i<VertexNum;i++)
89 {
90 Head[i].Vertex = i;
91 Head[i].Next = NULL;
92 }
93 printf("1.Undirected Graph\n");
94 printf("2.Directed Graph\n");
95 printf("Please choose : "); /* 选择有向图形或无向图形 */
96 scanf("%d",&Choose);
97 while (1)
98 {
99 printf("Please input the Edge's source : ");
100 scanf("%d",&Source);
101 if (Source == -1)
102 break;
103
104 printf("Please input the Edge's Destination : ");
105 scanf("%d",&Destination);
106
107 /* 错误: 超出范围 */
108 if (Source >= VertexNum || Destination >= VertexNum)
109 printf("@Error@ : Out of range!!\n");
110 else /* 调用建立邻接列表 */
111 {
112 New = (NextEdge) malloc(sizeof(struct Edge));
113 if (New != NULL) /* 配置成功 */
114 {
115 New->Vertex1 = Source; /* 邻近顶点 */
116 New->Vertex2 = Destination; /* 邻近顶点 */
117 New->Edge1 = NULL; /* 下一个邻接顶点指针 */
118 New->Edge2 = NULL; /* 下一个邻接顶点指针 */
119 Create_ML_Graph(Source,New);
120 if (Choose == 1)
121 Create_ML_Graph(Destination,New);
122 }
123 }
124 }
125 printf("##Graph##\n");
126 for (i=0;i<VertexNum;i++)
127 {
128 printf("Vertex[%d] : ",i);
129 Print_ML_Graph(&Head[i]); /* 调用输出多元邻接列表数据 */
130 }
131 }

```

运行结果:

```

C:\DS>ml_graph
1.Undirected Graph
2.Directed Graph
Please choose : 1
Please input the Edge's source : 1
Please input the Edge's Destination : 2
Please input the Edge's source : 1
Please input the Edge's Destination : 3

```

```

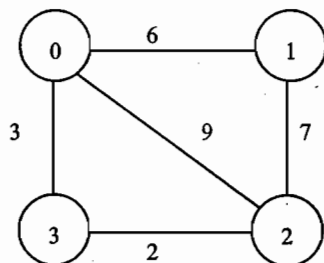
Please input the Edge's source : 1
Please input the Edge's Destination : 4
Please input the Edge's source : 2
Please input the Edge's Destination : 4
Please input the Edge's source : 3
Please input the Edge's Destination : 4
Please input the Edge's source : -1
##Graph##
Vertex[0] :
Vertex[1] : (1,2) (1,3) (1,4)
Vertex[2] : (1,2) (2,4)
Vertex[3] : (1,3) (3,4)
Vertex[4] : (1,4) (2,4) (3,4)
Vertex[5] :
C:\DS>

```

对于有向图形，读者可自行测试看看。

## 12.2.4 加权边的图形

我们之前所看到的图形都是在顶点与顶点间只存在边，但邻接边上并没有任何的符号。但是在生活中，如 7 座桥问题的图形，我们从图形中仅能了解到 4 个顶点和 7 个边，却无法知道各个桥的距离或其它信息。有时候，我们会在图形的边上加上一些数字，来表示一些数据，我们称之为“加权边的图形 (Weighted Edges Graph)”。如下图就是一个加权边的图形：



如果使用邻接数组来表示图形，则数组的内容值所代表的意义也必须做些调整，原来是以 1 来表示存在有边，以 0 表示不存在边。在加权边的图形上，邻接数组的内容值则是代表边的加权值，0 仍然表示不存在边。如上图的邻接数组为：

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 6 | 9 | 3 |
| 1 | 6 | 0 | 7 | 0 |
| 2 | 9 | 7 | 0 | 2 |
| 3 | 3 | 0 | 2 | 0 |

如果使用邻接列表或多重邻接列表来表示图形，则列表的节点必须添加一个记录加权值的字段。关于加权边图形的应用，之后我们会有更详细的介绍。

加权边的邻接列表节点结构：

| 加权值 | 邻接顶点 | 下一个邻接顶点 |
|-----|------|---------|
|-----|------|---------|

加权边的多重邻接列表节点结构:

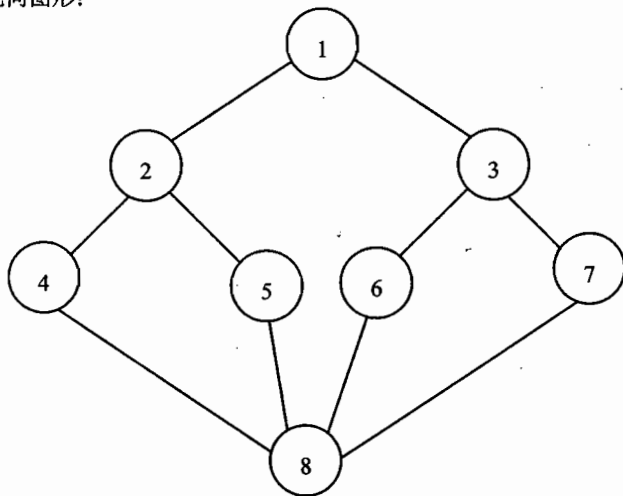
| Marked | Weight | Vertex1 | Vertex2 | Link1 | Link2 |
|--------|--------|---------|---------|-------|-------|
|--------|--------|---------|---------|-------|-------|

## 12.3 图形的查找

### 12.3.1 深度优先法

深度优先法(Depth-First-Search)是指在图形中, 如果以顶点  $v$  作为起始点开始查找, 我们从顶点  $v$  的邻接列表中选择一个未查找过的顶点  $w$ , 由顶点  $w$  继续进行深度优先法的查找, 每查找一个顶点, 便把该顶点存放在堆栈。直到查找到已经没有任何邻接列未遍历的顶点  $u$ , 此时回到取出堆栈中的顶点, 回到上一层顶点继续查找未遍历的顶点, 直到所有的顶点皆查找过为止。

对于下图的无向图形:



运作过程

1. 如果从顶点 1 开始深度优先搜索, 顶点 1 存入堆栈。
2. 顶点 1 的邻接顶点为顶点 2 和顶点 3, 选择顶点 2(也可以选择顶点 3)往下继续深度优先搜索。将顶点 2 存入堆栈。
3. 顶点 2 的邻接顶点为顶点 4 和顶点 5, 选择顶点 4 往下继续深度优先搜索。将顶点 4 存入堆栈。
4. 顶点 4 的邻接顶点为顶点 8, 顶点 8 已经是深度最深的。将顶点 8 存入堆栈。
5. 发现顶点 8 的邻接顶点为顶点 4、顶点 5、顶点 6 和顶点 7, 顶点 4 已经查找过, 选择顶点 5 继续深度优先搜索。将顶点 5 存入堆栈。
6. 发现顶点 5 的邻接顶点为顶点 2, 顶点 2 已经查找过,

堆栈内容

|   |
|---|
|   |
|   |
| 1 |

(1)

|   |
|---|
|   |
| 2 |
| 1 |

(2)

|   |
|---|
|   |
| 4 |
| 2 |
| 1 |

(3)

|   |
|---|
|   |
| 8 |
| 4 |
| 2 |
| 1 |

(4)

|   |
|---|
| 5 |
| 8 |
| 4 |
| 2 |
| 1 |

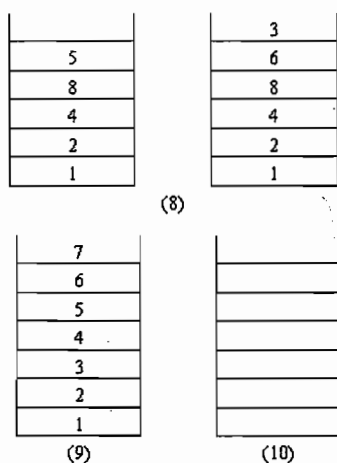
(5)

|   |
|---|
|   |
| 8 |
| 4 |
| 2 |
| 1 |

(6)

退回到顶点 8。将顶点 5 从堆栈取出。

7. 发现顶点 8 的邻接顶点为顶点 4、顶点 5、顶点 6 和顶点 7，顶点 4 和顶点 5 已经查找过，选择顶点 6 继续深度优先搜索。将顶点 6 存入堆栈。
8. 发现顶点 6 的邻接顶点为顶点 3，选择顶点 3 继续深度优先搜索。将顶点 3 存入堆栈。
9. 发现顶点 3 的邻接顶点为顶点 7，选择顶点 7 继续深度优先搜索。将顶点 7 存入堆栈。
10. 发现顶点 7 的邻接顶点皆查找完，取出堆栈中顶点。堆栈中所有顶点的邻接皆已查找(此时堆栈为空)。结束查找。



所以查找的顺序为：

顶点 1、顶点 2、顶点 4、顶点 8、顶点 5、顶点 6、顶点 3、顶点 7

因为深度优先搜索时，可选择同一深度的邻接顶点中一个继续进行邻接顶点的深度查找，所以深度优先搜索的顺序不是唯一的。

设计深度优先搜索程序时，我们可采用堆栈来存储未查找的邻接顶点或者采用递归来调用深度优先搜索函数，查找未曾查找过的顶点。

程序实例：

设计一个深度优先搜索法来查找上述图形的程序。

程序构思：

递归调用深度优先搜索法，往下一个邻接顶点查找，直到查找到邻接列表尾端为止。

程序源代码：

```
01 /* ===== Program Description ===== */
02 /* 程序名称: DFS.c */
03 /* 程序目的: 设计一个深度优先搜索法来查找图形的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define VertexNum 9 /* 定义顶点数 */
08
09 struct Node /* 声明图形顶点结构 */
10 {
11 int Vertex; /* 邻接顶点数据 */
12 struct Node *Next; /* 下一个邻接顶点 */
13 };
14 typedef struct Node *Graph; /* 定义图形结构 */
15 struct Node Head[VertexNum]; /* 顶点数组 */
16
17 int Visited[VertexNum]; /* 查找记录 */
18 /* ----- */
19 /* 深度优先搜索法 */
20 /* ----- */
```

```

21 void DFS(int Vertex)
22 {
23 Graph Pointer; /* 节点声明 */
24
25 Visited[Vertex] = 1; /* 已查找 */
26 printf("[%d]==>",Vertex);
27 Pointer = Head[Vertex].Next;
28
29 while (Pointer != NULL)
30 {
31 if (Visited[Pointer->Vertex] == 0)
32 DFS(Pointer->Vertex); /* 递归调用 */
33 Pointer = Pointer->Next; /* 下一个邻接点 */
34 }
35 }
36
37 /* ----- */
38 /* 建立邻接点至邻接列表内 */
39 /* ----- */
40 void Create_L_Graph(int Vertex1,int Vertex2)
41 {
42 Graph Pointer; /* 节点声明 */
43 Graph New; /* 新顶点声明 */
44
45 New = (Graph) malloc(sizeof(struct Node)); /* 配置内存 */
46 if (New != NULL) /* 配置成功 */
47 {
48 New->Vertex = Vertex2; /* 邻近顶点 */
49 New->Next = NULL; /* 下一个邻接顶点指针 */
50 /* Pointer 指针设为顶点数组之首节点 */
51 Pointer = &(Head[Vertex1]);
52
53 while (Pointer->Next != NULL)
54 Pointer = Pointer->Next; /* 往下一个节点 */
55
56 Pointer->Next = New; /* 串连在链接尾端 */
57 }
58 }
59
60 /* ----- */
61 /* 输出邻接列表内数据 */
62 /* ----- */
63 void Print_L_Graph(struct Node *Head)
64 {
65 Graph Pointer; /* 节点声明 */
66
67 Pointer = Head->Next; /* Pointer 指针设为首节点 */
68 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
69 {
70 printf("[%d]",Pointer->Vertex);
71 Pointer = Pointer->Next; /* 往下一个节点 */
72 }
73 printf("\n");
74 }
75
76 /* ----- */
77 /* 主程序 */
78 /* ----- */
79 void main ()
80 {
81 int i;

```



```

82 int Node[20][2] = { {1,2}, {2,1}, {1,3}, {3,1}, {2,4},
83 {4,2}, {2,5}, {5,2}, {3,6}, {6,3},
84 {3,7}, {7,3}, {4,8}, {8,4}, {5,8},
85 {8,5}, {6,8}, {8,6}, {7,8}, {8,7} };
86
87 for (i=0;i<VertexNum;i++)
88 {
89 Head[i].Vertex = i;
90 Head[i].Next = NULL;
91 }
92
93 for (i=0;i<VertexNum;i++)
94 Visited[i] = 0;
95
96 for (i=0;i<20;i++)
97 Create_L_Graph(Node[i][0],Node[i][1]);
98
99 printf("##Graph##\n");
100 for (i=1;i<VertexNum;i++)
101 {
102 printf("Vertex[%d] : ",i);
103 Print_L_Graph(&Head[i]); /* 调用输出邻接列表数据 */
104 }
105 printf("Depth-First-Search : \n");
106 printf("[BEGIN]==>");
107 DFS(1);
108 printf("[END]");
109 }

```

运行结果:

```

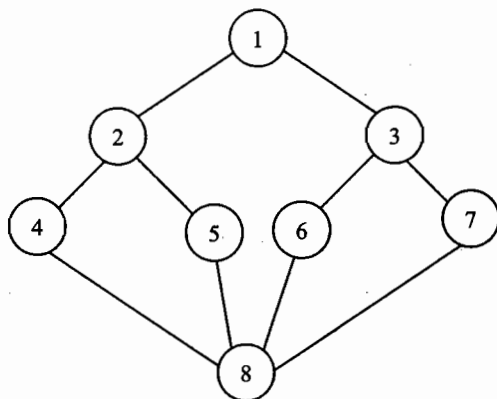
C:\DS>dfs
##Graph##
Vertex[1] : [2][3]
Vertex[2] : [1][4][5]
Vertex[3] : [1][6][7]
Vertex[4] : [2][8]
Vertex[5] : [2][8]
Vertex[6] : [3][8]
Vertex[7] : [3][8]
Vertex[8] : [4][5][6][7]
Depth-First-Search :
[BEGIN]==>[1]==>[2]==>[4]==>[8]==>[5]==>[6]==>[3]==>[7]==>[END]
C:\DS>

```

### 12.3.2 广度优先法

广度优先法(Bradth-First-Search)是指在图形中, 如果以顶点  $v$  作为起始点开始查找, 我们从顶点  $v$  的邻接列表中选择一个未查找过的顶点  $w$ , 将顶点  $v$  的所有邻接顶点查找过后, 再继续对顶点  $w$  的所有邻接顶点进行广度优先法的查找, 然后再继续查找顶点  $v$  的下一个邻接顶点的所有邻接顶点, 重复进行广度优先搜索, 直到所有的邻接顶点皆查找过为止。通常是使用队列来存储邻接顶点, 每查找一个邻接顶点便把所有的邻接顶点存入队列中, 直到队列空了才结束广度优先搜索。

对于下图的无向图形:



1. 如果从顶点 4 开始广度优先搜索, 将顶点 4 存入队列中。

|   |  |  |  |  |  |
|---|--|--|--|--|--|
| 4 |  |  |  |  |  |
|---|--|--|--|--|--|

2. 查找顶点 4, 将顶点 4 的邻接顶点存入队列中, 将顶点 4 从队列中取出。

|   |   |  |  |  |  |
|---|---|--|--|--|--|
| 2 | 8 |  |  |  |  |
|---|---|--|--|--|--|

3. 查找顶点 2, 将顶点 2 的邻接顶点存入队列中, 邻接顶点 4 已查找过, 不必存入队列中。将顶点 2 从队列中取出。

|   |   |   |  |  |  |
|---|---|---|--|--|--|
| 8 | 1 | 5 |  |  |  |
|---|---|---|--|--|--|

4. 查找顶点 8, 将顶点 8 的邻接顶点存入队列中, 邻接顶点 4 已查找过和邻接顶点 5 已在队列中, 所以不必存入队列, 将顶点 8 从队列中取出。

|   |   |   |   |  |  |
|---|---|---|---|--|--|
| 1 | 5 | 6 | 7 |  |  |
|---|---|---|---|--|--|

5. 查找顶点 1, 将顶点 1 的邻接顶点存入队列中, 邻接顶点 2 已查找过, 不必存入队列中。将顶点 1 从队列中取出。

|   |   |   |   |  |  |
|---|---|---|---|--|--|
| 5 | 6 | 7 | 3 |  |  |
|---|---|---|---|--|--|

6. 查找顶点 5, 将顶点 5 的邻接顶点存入队列中, 邻接顶点 2 和邻接顶点 8 已查找过, 不必存入队列中, 将顶点 5 从队列中取出。

|   |   |   |  |  |  |
|---|---|---|--|--|--|
| 6 | 7 | 3 |  |  |  |
|---|---|---|--|--|--|

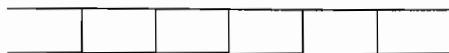
7. 查找顶点 6, 将顶点 6 的邻接顶点存入队列中, 邻接顶点 8 已查找过和邻接顶点 3 已在队列中, 所以不必存入队列。将顶点 6 从队列中取出。

|   |   |  |  |  |  |
|---|---|--|--|--|--|
| 7 | 3 |  |  |  |  |
|---|---|--|--|--|--|

8. 查找顶点 7, 将顶点 7 的邻接顶点存入队列中, 邻接顶点 8 已查找过和邻接顶点 3 已在队列中, 不必存入队列中。将顶点 7 从队列中取出。

|   |  |  |  |  |  |
|---|--|--|--|--|--|
| 3 |  |  |  |  |  |
|---|--|--|--|--|--|

9. 查找顶点 3, 顶点 3 的邻接顶点皆已查找过, 不必存入队列。将顶点 3 从队列中取出。此时队列为空, 结束查找。



所以查找的顺序为:

顶点 4、顶点 2、顶点 8、顶点 1、顶点 5、顶点 6、顶点 7、顶点 3

因为广度优先搜索时, 同一广度的邻接顶点, 可选择其中一个继续进行邻接顶点的广度查找, 所广度优先搜索的顺序也不是唯一的。

程序实例:

设计一个广度优先搜索法来查找上述图形的程序。

程序构思:

查找顶点时, 先将该顶点的邻接顶点皆存入队列中。

(关于队列的运用, 之前的章节已有介绍, 在此不再做重复)

如果邻接顶点已存在放队列中或已查找, 则不存入队列中, 直到队列为空才结束查找工作。

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: BFS.c */
03 /* 程序目的: 设计一个广度优先搜索法来查找图形的程序。 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define VertexNum 9 /* 定义顶点数 */
08 #define QueueMax 10
09
10 struct Node /* 声明图形顶点结构 */
11 {
12 int Vertex; /* 邻接顶点数据 */
13 struct Node *Next; /* 下一个邻接顶点 */
14 };
15 typedef struct Node *Graph; /* 定义图形结构 */
16 struct Node Head[VertexNum]; /* 顶点数组 */
17
18 int Queue[QueueMax];
19 int Front = -1;
20 int Rear = -1;
21
22 int Visited[VertexNum]; /* 查找记录 */
23 /* ----- */
24 /* 队列的存入 */
25 /* ----- */
26 int Enqueue(int Vertex)
27 {
28 if (Rear >= QueueMax) /* 队列已满 */
29 return -1;
30 else
31 {
32 Rear++; /* 队列尾端指针后移 */
33 Queue[Rear] = Vertex; /* 将值存入队列中 */

```

```

34 return 1;
35 }
36 }
37
38 /* ----- */
39 /* 队列的取出 */
40 /* ----- */
41 int Dequeue()
42 {
43 if (Front == Rear) /* 队列已空 */
44 return -1;
45 else
46 {
47 Front++; /* 队头指针后移 */
48 return Queue[Front];
49 }
50 }
51
52 /* ----- */
53 /* 广度优先搜索法 */
54 /* ----- */
55 void BFS(int Vertex)
56 {
57 Graph Pointer; /* 节点声明 */
58
59 Enqueue(Vertex); /* 存入队列中 */
60 Visited[Vertex] = 1; /* 已查找 */
61 printf("[%d]==>",Vertex);
62
63 while (Front != Rear) /* 队列为空时, 结束循环 */
64 {
65 Vertex = Dequeue();
66 Pointer = Head[Vertex].Next;
67 while (Pointer != NULL) /* 读入邻接列表所有顶点 */
68 {
69 if (Visited[Pointer->Vertex] == 0)
70 {
71 Enqueue(Pointer->Vertex); /* 存入队列中 */
72 Visited[Pointer->Vertex] = 1; /* 已查找过的顶点 */
73 printf("[%d]==>",Pointer->Vertex);
74 }
75 Pointer = Pointer->Next; /* 下一个邻接点 */
76 }
77 }
78 }
79
80 /* ----- */
81 /* 建立邻接顶点至邻接列表内 */
82 /* ----- */
83 void Create_L_Graph(int Vertex1,int Vertex2)
84 {
85 Graph Pointer; /* 节点声明 */
86 Graph New; /* 新顶点声明 */
87
88 New = (Graph) malloc(sizeof(struct Node)); /* 配置内存 */
89 if (New != NULL) /* 配置成功 */
90 {
91 New->Vertex = Vertex2; /* 邻近顶点 */
92 New->Next = NULL; /* 下一个邻接顶点指针 */
93 /* Pointer 指针设为顶点数组之首节点 */
94 Pointer = &(Head[Vertex1]);

```

```

95
96 while (Pointer->Next != NULL)
97 Pointer = Pointer->Next; /* 往下一个节点 */
98
99 Pointer->Next = New; /* 串连在链接尾端 */
100 }
101 }
102
103 /* ----- */
104 /* 输出邻接列表内数据 */
105 /* ----- */
106 void Print_L_Graph(struct Node *Head)
107 {
108 Graph Pointer; /* 节点声明 */
109
110 Pointer = Head->Next; /* Pointer 指针设为首节点 */
111 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
112 {
113 printf("[%d]",Pointer->Vertex);
114 Pointer = Pointer->Next; /* 往下一个节点 */
115 }
116 printf("\n");
117 }
118
119 /* ----- */
120 /* 主程序 */
121 /* ----- */
122 void main ()
123 {
124 int i;
125 int Node[20][2] = { {1,2}, {2,1}, {1,3}, {3,1}, {2,4},
126 {4,2}, {2,5}, {5,2}, {3,6}, {6,3},
127 {3,7}, {7,3}, {4,8}, {8,4}, {5,8},
128 {8,5}, {6,8}, {8,6}, {7,8}, {8,7} };
129
130 for (i=0;i<VertexNum;i++)
131 {
132 Head[i].Vertex = i;
133 Head[i].Next = NULL;
134 }
135
136 for (i=0;i<VertexNum;i++)
137 Visited[i] = 0;
138
139 for (i=0;i<20;i++)
140 Create_L_Graph(Node[i][0],Node[i][1]);
141
142 printf("##Graph##\n");
143 for (i=1;i<VertexNum;i++)
144 {
145 printf("Vertex[%d] : ",i);
146 Print_L_Graph(&Head[i]); /* 调用输出邻接列表数据 */
147 }
148 printf("Bradth-First-Search : \n");
149 printf("[BEGIN]==>");
150 BFS(4);
151 printf("[END]");
152 }

```

运行结果:

```
C:\DS>bfs
##Graph##
Vertex[1] : [2][3]
Vertex[2] : [1][4][5]
Vertex[3] : [1][6][7]
Vertex[4] : [2][8]
Vertex[5] : [2][8]
Vertex[6] : [3][8]
Vertex[7] : [3][8]
Vertex[8] : [4][5][6][7]
Brath-First-Search :
[BEGIN]==>[4]==>[2]==>[8]==>[1]==>[5]==>[6]==>[7]==>[3]==>[END]
C:\DS>
```

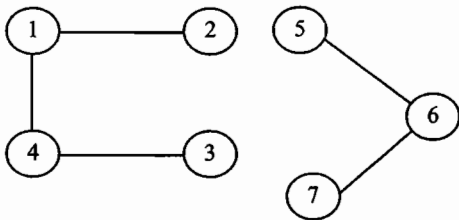
读者可以自行用不同的起始顶点, 测试深度优先搜索或广度优先搜索, 相信对于读者了解图形的深度及广度优先搜索会有很大的助益。

### 12.3.3 连通组件

连通图形的判断, 通常只要建立图形之后, 只要从第一个顶点做深度优先搜索或广度优先搜索, 之后看看是不是所有的顶点都查找过, 如果所有的顶点都查找过, 则表示这个图形是连通的图形, 否则表示这个图形不是连通的图形。

连通组件的判断, 也只需要重复的对未查找过的顶点做深度优先搜索或广度优先搜索, 输出边, 即可输出图形的连通组件。

如下图的无向图形:



如果用深度优先搜索, 从顶点 1 开始查找, 会发现顶点 5、顶点 6、顶点 7 没有被查找过。所以这个图形并不是连通的图形。

如果想要找出连通组件, 则从顶点 1 开始查找, 则深度优先搜索的顺序为: 顶点 1、顶点 2、顶点 4、顶点 3, 则顶点 1、顶点 2、顶点 4、顶点 3 为一个连通组件。未查找的顶点 5, 再做深度优先搜索, 其顺序为: 顶点 5、顶点 6、顶点 7, 则顶点 5、顶点 6、顶点 7 又为另一个连通组件。

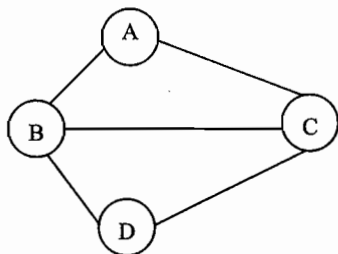
## 12.4 生成树问题

### 12.4.1 生成树

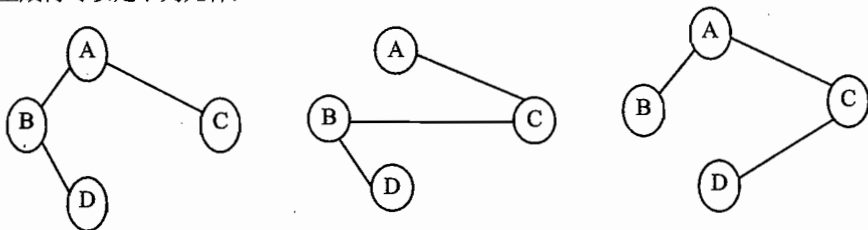
在图形中, 如果有  $N$  个顶点, 则至少要有  $N-1$  个边才能将  $N$  个顶点给相连接起来, 形成连通图形, 这

种包含着  $N-1$  个边的连通图形，我们称之为生成树(Spanning Tree)。

如无向图形：

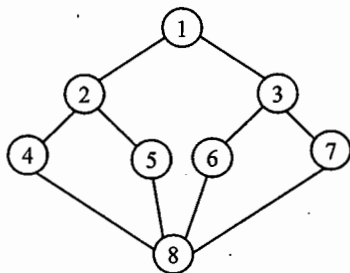


其生成树可以是下列几种：

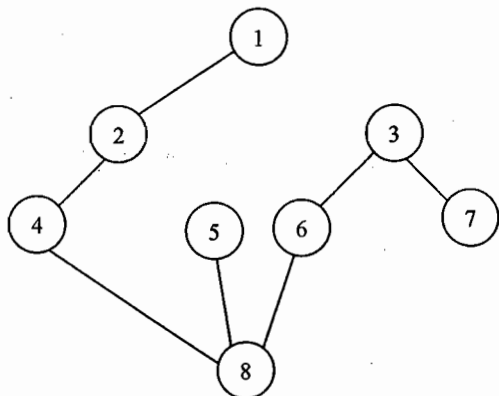


我们可以使用深度优先搜索或广度优先搜索来查找图形中的所有顶点，在查找的过程中，会把图形的边区分成两类，一类是查找树边，一类是非查找树边，查找边是指在查找过程中会用到的边，而非查找树边则是查找后未用到的边。所以我们可以运用深度优先搜索或广度优先搜索所产生的查找树边来建立生成树。我们称之为深度优先搜索的生成树(DFS Spanning Tree)或广度优先搜索的生成树(BFS Spanning Tree)。

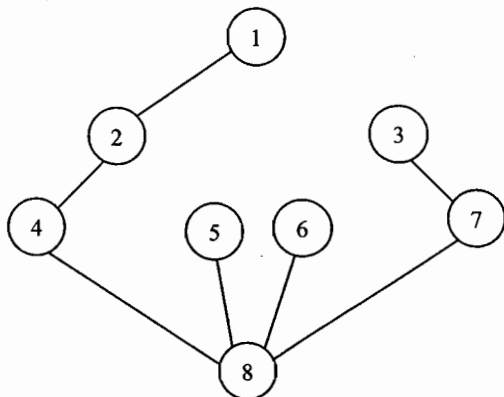
如无向图形：



以顶点 1 开始的深度优先搜索，所产生的生成树为：

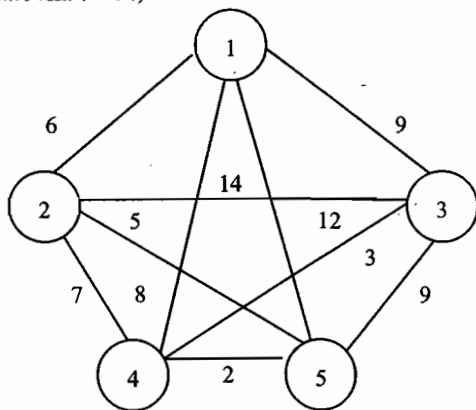


以顶点 4 开始的广度优先搜索，所产生的生成树为：



### 12.4.2 最小生成树

我们曾介绍过加权边的图形，在一个加权边图形的所有生成树中，加权值总和最小的生成树，我们称之为“最小生成树(Minimal Spanning Tree)”。例如：我们要建设局域网络，而其局域网络上有 5 栋大楼，为了能运用网络来连接各栋大楼，所以我们必须在大楼与大楼间使用网络线，经过初步的测量之后，我们得到下列这张图(边上的加权值为距离)：



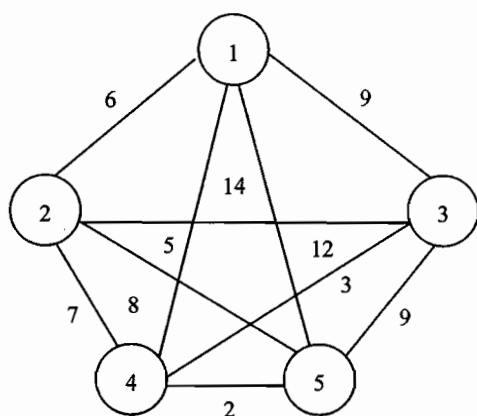
如果能找出这张图形的最小生成树，即可以用最少的网络线串连 5 栋大楼，完成局域网络的建设。

### 12.4.3 Kruskal 算法

接下来我们要介绍两种演算来找出最小生成树。首先我们要谈的是 Kruskal 算法。

Kruskal 算法是根据边的加权值以递增的方式，依次找出加权值最低的边来建最小生成树，而且规定：每次添加的边不能造成生成树有回路，直到找到  $N-1$  个边为止。我们以之前的网络建设图来作为例子：

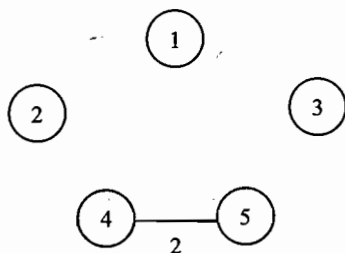




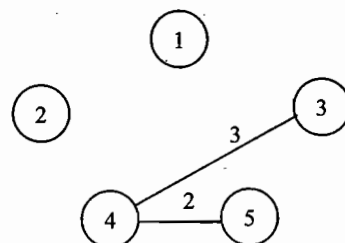
首先，我们将图形中的所有边递增排序：

| 邻接边 | (4,5) | (3,4) | (1,4) | (1,3) | (1,2) | (2,4) | (2,5) | (3,5) | (1,5) | (2,3) |
|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 加权值 | 2     | 3     | 5     | 6     | 7     | 8     | 8     | 9     | 12    | 14    |

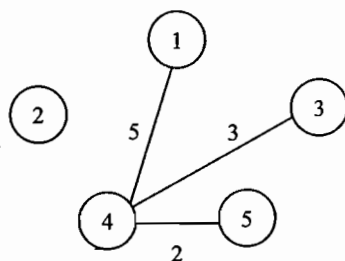
步骤 1：将(4,5)的边加到生成树中。



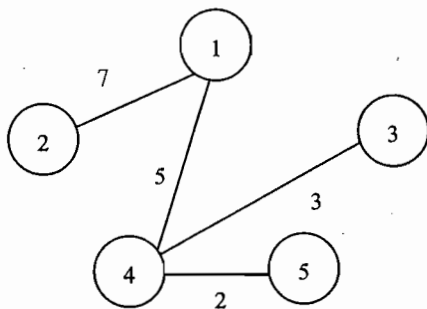
步骤 2：将(3,4)的边加到生成树中。



步骤 3：将(1,4)的边加到生成树中。



步骤 4: (1,3)的边会形成回路, 所以不能加到生成树中。将(1,2)的边加到生成树中。  
共有 5 个顶点, 生成树已达 4 个边, 生成树建立完成。



建立好的生成树的加权值总和为 17。

#### 程序实例:

设计一个利用 Kruskal 算法找出生成树的程序。

#### 程序构思:

先将所有的邻接边依加权值递增用链表链接起来。节点结构为:

| Vertex1 | Vertex2 | Weight | Next |
|---------|---------|--------|------|
|---------|---------|--------|------|

另声明一个一维数组 Tree, 表示现在生成树集合中的顶点, 用于判断是否产生回路。用 1 表示生成树的顶点, 0 表示非生成树的顶点, 如果新加入的邻接边中两个顶点皆在生成树集合中, 表示会产生回路。

利用 Kruskal 算法之前建立的链表来加入生成树的邻接边, 直到产生  $NN-1$  个边为止。

最后将结果输出。

#### 程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: kruskal.c */
03 /* 程序目的: 设计一个利用 Kruskal 算法找出生成树的程序 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define Max 10
08 #define VertexNum 5
09
10 struct List /* 节点结构声明 */
11 {
12 int Vertex1;
13 int Vertex2;
14 int Weight;
15 struct List *Next;
16 };
17 typedef struct List Node;
18 typedef Node *Edge; /* 定义邻接边的节点 */

```

```

19
20 int Edges[10][3] = /* 输入数据 */
21 { {1,2,7}, {1,3,6}, {1,4,5}, {1,5,12}, {2,3,14},
22 {2,4,8}, {2,5,8}, {3,4,3}, {3,5,9}, {4,5,2} };
23
24 int Visited[VertexNum]; /* 查找记录 */
25
26 /* ----- */
27 /* Kruskal 算法 */
28 /* ----- */
29 void Kruskal(Edge Head)
30 {
31 Edge Pointer; /* 节点声明 */
32 int EdgeNum = 0; /* 已连结边的数目 */
33 int Weight = 0;
34
35 Pointer = Head;
36
37 /* 当边的数目为顶点的数目减1时, 结束循环 */
38 while (EdgeNum != (VertexNum - 1))
39 {
40 /* 有一顶点不在生成树中时 */
41 if (Visited[Pointer->Vertex1] == 0
42 || Visited[Pointer->Vertex2] == 0)
43 {
44
45 printf("=>[%d,%d]", Pointer->Vertex1, Pointer->Vertex2);
46 printf("(%d)", Pointer->Weight);
47 Weight += Pointer->Weight;
48 EdgeNum++; /* 边数加1 */
49 Visited[Pointer->Vertex1] = 1; /* 设为已查找 */
50 Visited[Pointer->Vertex2] = 1; /* 设为已查找 */
51 }
52 Pointer = Pointer->Next; /* 往下一个节点 */
53
54 if (Pointer == NULL) /* 已无边时 */
55 {
56 printf("No Spanning Tree \n");
57 break;
58 }
59 }
60 printf("\nTotal weight = %d\n", Weight); /* 输出加权值总和 */
61 }
62
63 /* ----- */
64 /* 输出链表数据 */
65 /* ----- */
66 void Print_Edge(Edge Head)
67 {
68 Edge Pointer; /* 节点声明 */
69 Pointer = Head; /* Pointer 指针设为首节点 */
70 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
71 {
72 printf("[%d,%d]", Pointer->Vertex1, Pointer->Vertex2);
73 printf("(%d)", Pointer->Weight); /* 输出加权值 */
74 Pointer = Pointer->Next; /* 往下一个节点 */
75 }
76 printf("\n");
77 }
78
79 /* ----- */

```

```

80 /* 递增插入邻接边至链表内 */
81 /* ----- */
82 Edge Insert_Edge(Edge Head, Edge New)
83 {
84 Edge Pointer; /* 节点声明 */
85
86 Pointer = Head; /* Pointer 指针设为首节点 */
87
88 while (1)
89 {
90 if (New->Weight < Head->Weight) /* 新的加权值比首节点小 */
91 {
92 New->Next = Head; /* 插入在首节点之前 */
93 Head = New;
94 Break;
95 }
96
97 /* 插入在链表中间或尾端 */
98 if (New->Weight >= Pointer->Weight
99 && New->Weight < Pointer->Next->Weight)
100 {
101 New->Next = Pointer->Next;
102 Pointer->Next = New;
103 Break;
104 }
105 Pointer = Pointer->Next; /* 往下一个节点 */
106 }
107 return Head;
108 }
109 /* ----- */
110 /* 释放链表 */
111 /* ----- */
112 void Free_Edge(Edge Head)
113 {
114 Edge Pointer; /* 节点声明 */
115
116 while (Head != NULL) /* 当节点为 NULL 结束循环 */
117 {
118 Pointer = Head;
119 Head = Head->Next; /* 往下一个节点 */
120 free(Pointer);
121 }
122 }
123
124 /* ----- */
125 /* 建立链表 */
126 /* ----- */
127 Edge Create_Edge(Edge Head)
128 {
129 Edge New; /* 节点声明 */
130 Edge Pointer; /* 节点声明 */
131 int i;
132
133 Head = (Edge) malloc(sizeof(Node)); /* 内存配置 */
134
135 if (Head == NULL)
136 printf("Memory allocate Failure!!\n"); /* 内存配置失败 */
137 else
138 {
139 {
140 Head->Vertex1 = Edges[0][0];

```

```

141 Head->Vertex2 = Edges[0][1];
142 Head->Weight = Edges[0][2]; /* 定义节点加权值 */
143 Head->Next = NULL;
144
145 for (i=1; i<10; i++)
146 {
147 New = (Edge) malloc(sizeof(Node));
148
149 if (New != NULL)
150 {
151 New->Vertex1 = Edges[i][0];
152 New->Vertex2 = Edges[i][1];
153 New->Weight = Edges[i][2]; /* 定义节点加权值 */
154 New->Next = NULL;
155
156 Head = Insert_Edge(Head, New); /* 则插入新节点 */
157 }
158 }
159 return Head;
160 }
161
162 /* ----- */
163 /* 主程序 */
164 /* ----- */
165 void main ()
166 {
167 Edge Head; /* 节点声明 */
168 int i;
169
170 for (i=0; i<VertexNum; i++) /* 清除查找记录 */
171 Visited[i] = 0;
172
173 Head = Create_Edge(Head); /* 调用建立链表 */
174
175 if (Head != NULL)
176 {
177 printf("Kruskal Algorithm : \n");
178 printf("First Step : Sorting \n");
179 Print_Edge(Head);
180 printf("Second Step : Find Minimal Spanning Tree. \n");
181
182 Kruskal(Head); /* 调用 kruskal 算法 */
183
184 Free_Edge(Head); /* 调用释放链表 */
185 }
186 }
187

```

运行结果:

```

C:\DS>kruskal
Kruskal Algorithm :
First Step : Sorting
[4,5] (2) [3,4] (3) [1,4] (5) [1,3] (6) [1,2] (7) [2,4] (8)
[2,5] (8) [3,5] (9) [1,5] (12) [2,3] (14)
Second Step : Find Minimal Spanning Tree.
==>[4,5] (2)==>[3,4] (3)==>[1,4] (5)==>[1,2] (7)
Total weight = 17

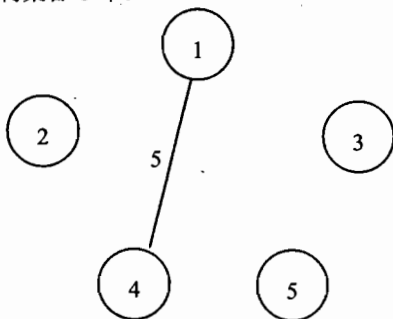
```

C:\DS&gt;

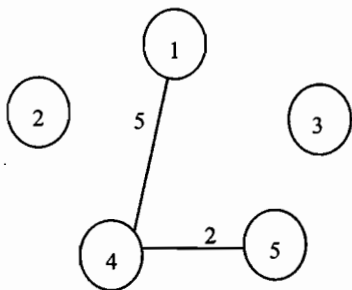
### 12.4.4 Prims 算法

Prims 算法是以每次加入一个的邻接边来建立最小生成树, 直到找到  $N-1$  个边为止。Prims 的规则是以开始时生成树的集合(集合  $U$ )为起始的顶点, 然后找出与生成树集合邻接的边(集合  $V$ )中, 加权值最小的边建立来生成树, 为了确定新加入的边不会造成回路, 所以每一个新加入的边, 只允许有一个顶点在生成树集合中, 重复执行此步骤, 直到找到  $N-1$  个边为止。我们以之前的网络建设图来作为例子(假设我们以顶点 1 开始):

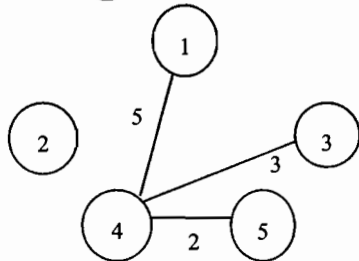
步骤 1: 生成树集合  $U = \{1\}$ 、邻接边集合  $V = \{(1,2), (1,3), (1,4), (1,5)\}$ 。邻接边集合中最小的边为  $(1,4)$ , 将  $(1,4)$  加入生成树集合  $U$  中。



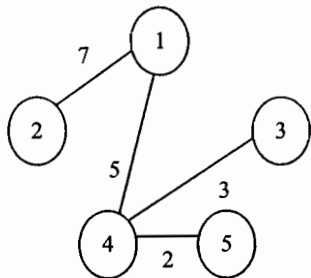
步骤 2: 生成树集合  $U = \{(1,4)\}$ 、邻接边集合  $V = \{(1,2), (1,3), (1,4), (1,5), (4,1), (4,2), (4,3), (4,5)\}$ , 其中  $(1,4)$  和  $(4,1)$  中有两个顶点皆属于生成树集合, 故不合题意。其余的邻接边集合中最小的边为  $(4,5)$ , 将  $(4,5)$  加入生成树集合  $U$  中。



步骤 3: 生成树集合  $U = \{(1,4), (4,5)\}$ 、邻接边集合  $V = \{(1,2), (1,3), (1,4), (1,5), (4,1), (4,2), (4,3), (4,5), (3,1), (3,2), (3,4), (3,5)\}$ , 其中  $(1,3)$ 、 $(1,4)$ 、 $(4,1)$ 、 $(4,3)$ 、 $(3,1)$ 、 $(3,4)$  中有两个顶点皆属于生成树集合, 故不合题意。其余的邻接边集合中最小的边为  $(4,3)$ , 将  $(4,3)$  加入生成树集合  $U$  中。



步骤 4: 生成树集合  $U = \{(1,4), (4,3), (4,5)\}$ 、邻接边集合  $V = \{(1,2), (1,3), (1,4), (1,5), (4,1), (4,2), (4,3), (4,5), (3,1), (3,2), (3,4), (3,5), (5,1), (5,2), (5,3), (5,4)\}$ , 其中  $(1,3)$ 、 $(1,4)$ 、 $(1,5)$ 、 $(4,1)$ 、 $(4,3)$ 、 $(4,5)$ 、 $(3,1)$ 、 $(3,4)$ 、 $(3,5)$ 、 $(5,1)$ 、 $(5,3)$ 、 $(5,4)$  中有两个顶点皆属于生成树集合, 故不合题意。其余的邻接边集合中最小的边为  $(1,2)$ , 将  $(1,2)$  加入生成树集合  $U$  中。



设计一个利用 Prims 算法找出生成树的程序。

程序构思：

先将所有的邻接边用链表链接起来。节点结构为：

| Marked | Vertex1 | Vertex2 | Weight | Next |
|--------|---------|---------|--------|------|
|--------|---------|---------|--------|------|

(Marked 表示此边是否已用过)

另声明一个一维数组 *Tree*，表示现在生成树集合中的顶点，用于判断是否产生回路。用 1 表示生成树的顶点，0 表示非生成树的顶点，如果新加入的邻接边中两个顶点皆在生成树集合中，表示会产生回路。

利用 Prims 算法将之前建立的链表来加入生成树的邻接边，直到产生  $N-1$  个边为止。

最后将结果输出。

程序源代码：

```

01 /* ===== Program Description ===== */
02 /* 程序名称: prims.c - */
03 /* 程序目的: 设计一个利用 Prims 算法找出生成树的程序 */
04 /* Written By Kuo-Yu Huang. (WANT Studio.) */
05 /* ===== */
06 #include <stdlib.h>
07 #define Max 10
08 #define VertexNum 5
09
10 struct List /* 节点结构声明 */
11 {
12 int Marked; /* 查找标记 */
13 int Vertex1; /* 顶点声明 */
14 int Vertex2; /* 顶点声明 */
15 int Weight; /* 加权值 */
16 struct List *Next;
17 };
18 typedef struct List Node;
19 typedef Node *Edge; /* 定义邻接边的节点 */
20
21 int Edges[10][3] = /* 输入数据 */
22 {
23 {1,2,7}, {1,3,6}, {1,4,5}, {1,5,12}, {2,3,14},
24 {2,4,8}, {2,5,8}, {3,4,3}, {3,5,9}, {4,5,2} };
25
26 int Visited[VertexNum+1]; /* 查找记录 */
27
28 /* ----- */
29 /* Prims 算法 ----- */
30 void Prims(Edge Head,int Index)
31 {
32 Edge Pointer; /* 节点声明 */
33 Edge MinEdge;
34 int EdgeNum = 0; /* 已连结边的数目 */
35 int Weight = 0; /* 累计加权值 */

```

```

36 int i;
37
38 Visited[Index] = 1; /* 设置已查找过 */
39
40 /* 当边的数目为顶点的数目减1时, 结束循环 */
41 while (EdgeNum != (VertexNum - 1))
42 {
43 MinEdge->Weight = 999; /* 将最小边的加权值设到最大 */
44
45 for (i=1;i<=VertexNum;i++) /* 判断未建立的邻接顶点 */
46
47 {
48 Pointer = Head;
49 If (Visited[i] == 1) /* 已存在生成树集合的顶点 */
50 {
51 while (Pointer->Marked == 1) /* 往下一个未建立的邻接边 */
52 Pointer = Pointer->Next;
53 if (MinEdge->Weight > Pointer->Weight)
54 MinEdge = Pointer; /* 找出加权值最小的邻接边 */
55
56 while (Pointer != NULL)
57
58 {
59 /* 如果两顶点皆存在生成树集合中, 表示是已查找过的边 */
60
61 if (Visited[Pointer->Vertex1] == 1
62 && Visited[Pointer->Vertex2] == 1)
63 Pointer->Marked = 1;
64 /* 找出加权值最小的邻接边 */
65 if (MinEdge->Weight > Pointer->Weight
66 && Pointer->Marked == 0
67 && (Pointer->Vertex1 == i
68 || Pointer->Vertex2 == i))
69 MinEdge = Pointer;
70 Pointer = Pointer->Next;
71 }
72 }
73 Visited[MinEdge->Vertex1] = 1; /* 将顶点1设为存在生成树集合中 */
74
75 Visited[MinEdge->Vertex2] = 1; /* 将顶点2设为存在生成树集合中 */
76
77 EdgeNum++; /* 生成树的边数加1 */
78 Weight += MinEdge->Weight; /* 累计加权值 */
79 Printf("[%d,%d]",MinEdge->Vertex1,MinEdge->Vertex2);
80 Printf("(%d)",MinEdge->Weight);
81 }
82 printf("\nTotal weight = %d\n",Weight); /* 输出加权值总和 */
83 }
84
85 /* ----- */
86 /* 释放链表 ----- */
87 /* ----- */
88 void Free_Edge(Edge Head)
89 {
90 Edge Pointer; /* 节点声明 */
91
92 while (Head != NULL) /* 当节点为NULL结束循环 */
93 {
94 Pointer = Head;
95 Head = Head->Next; /* 往下一个节点 */
96 free(Pointer);

```



```

97 }
98 }
99
100 /* ----- */
101 /* 输出链表数据 */
102 /* ----- */
103 void Print_Edge(Edge Head)
104 {
105 Edge Pointer; /* 节点声明 */
106
107 Pointer = Head; /* Pointer 指针设为首节点 */
108 while (Pointer != NULL) /* 当节点为 NULL 结束循环 */
109 {
110 printf("[%d,%d]",Pointer->Vertex1,Pointer->Vertex2);
111 printf("(%d)",Pointer->Weight); /* 输出加权值 */
112 Pointer = Pointer->Next; /* 往下一个节点 */
113 }
114 printf("\n");
115 }
116
117 /* ----- */
118 /* 插入邻接边至链表中 */
119 /* ----- */
120 Edge Insert_Edge(Edge Head,Edge New)
121 {
122 Edge Pointer; /* 节点声明 */
123
124 Pointer = Head; /* Pointer 指针设为首节点 */
125
126 while (Pointer->Next != NULL) /* 插入在链表尾端 */
127 Pointer = Pointer->Next;
128 Pointer->Next = New;
129
130 return Head;
131 }
132
133 /* ----- */
134 /* 建立链表 */
135 /* ----- */
136 Edge Create_Edge(Edge Head)
137 {
138 Edge New; /* 节点声明 */
139 Edge Pointer; /* 节点声明 */
140 int I;
141
142 Head = (Edge) malloc(sizeof(Node)); /* 内存配置 */
143
144 if (Head == NULL)
145 printf("Memory allocate Failure!!\n"); /* 内存配置失败 */
146 else
147 {
148 Head->Marked = 0;
149 Head->Vertex1 = Edges[0][0];
150 Head->Vertex2 = Edges[0][1];
151 Head->Weight = Edges[0][2]; /* 定义节点加权值 */
152 Head->Next = NULL;
153
154 for (i=1;i<10;i++)
155 {
156 New = (Edge) malloc(sizeof(Node));

```

```

158
159 If (New != NULL)
160 {
161 New->Marked = 0;
162 New->Vertex1 = Edges[i][0];
163 New->Vertex2 = Edges[i][1];
164 New->Weight = Edges[i][2]; /* 定义节点加权值 */
165 New->Next = NULL;
166
167 Head = Insert_Edge(Head,New); /* 则插入新节点 */
168 }
169 }
170 }
171 return Head;
172 }
173 /* ----- */
174 /* 主程序 */
175 /* ----- */
176 void main ()
177 {
178 Edge Head; /* 节点声明 */
179 int I;
180
181 for (I=1;I<=VertexNum;I++) /* 清除查找记录 */
182 Visited[i] = 0;
183
184 Head = Create_Edge(Head); /* 调用建立链表 */
185 Print_Edge(Head);
186
187 if (Head != NULL)
188 {
189 printf("Prims Algorithm : \n");
190 printf("Start from Vertex 1.\n");
191 printf("Find Minimal Spanning Tree.\n");
192
193 Prims(Head,1); /* 调用 Prims 算法 */
194 free(Head); /* 调用释放链表 */
195 }
196 }
197 }
198

```

运行结果:

```

C:\DS>prims
[1,2] (7) [1,3] (6) [1,4] (5) [1,5] (12) [2,3] (14)
[2,4] (8) [2,5] (8) [3,4] (3) [3,5] (9) [4,5] (2)
Prims Algorithm :
Start from Vertex 1.
Find Minimal Spanning Tree.
[1,4] (5) [4,5] (2) [3,4] (3) [1,2] (7)
Total weight = 17
C:\DS>

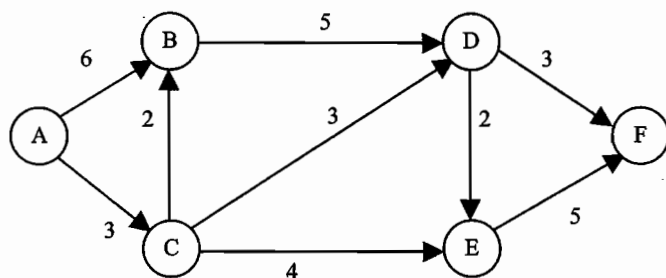
```

## 12.5 最短路径问题

在日常生活中, 我们如果需要常常往返 A 城市和 B 城市间, 我们最希望知道的可能是从 A 城市到 B 城市间的众多路径中, 那一条路径的路途最短。在本节, 我们所要讨论的正是如何解这类的最短路径问

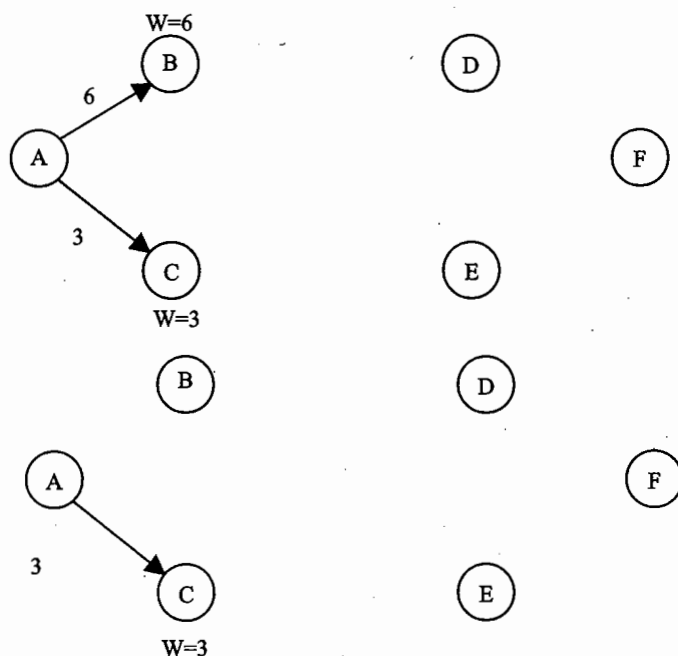
题。

如果我们把城市间的众多路径画成图形。假设我们有一张城市的地图，以图形表示如下：

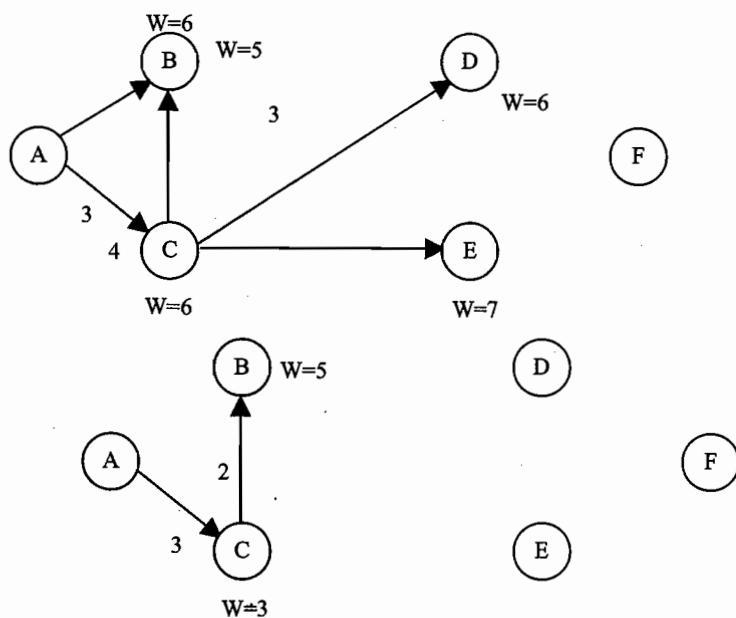


图形中，邻接边上的加权值表示城市间的距离。那么我们该如何找出城市 A 到各个城市的最短路径呢？我们采用一种叫 Dijkstra 的算法。接下来，我们先以图标来说明 Dijkstra 算法。从上图我们得知我们想从城市 A 出发。

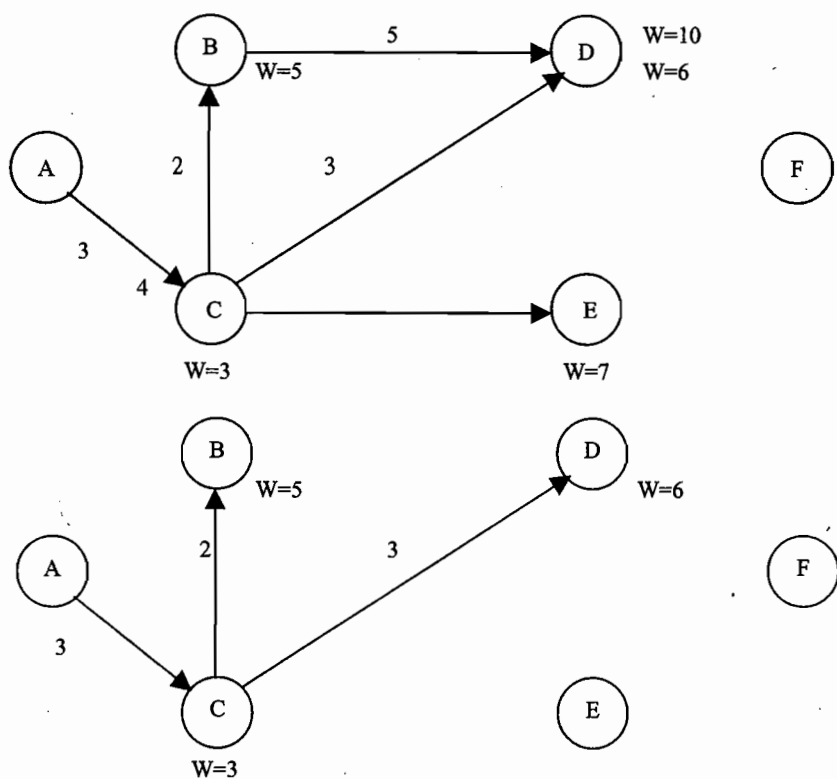
首先，先找出和城市 A 邻近，且有路径的城市，城市 B 和城市 C。从图中我们知道从城市 A 到城市 B 的路径为 6，城市 A 到城市 C 的路径为 3。我们在图形上标记现在的路径总和 W。此时从城市 A 到城市 B 的总和路径为 6、从城市 A 到城市 C 的总和路径为 3。



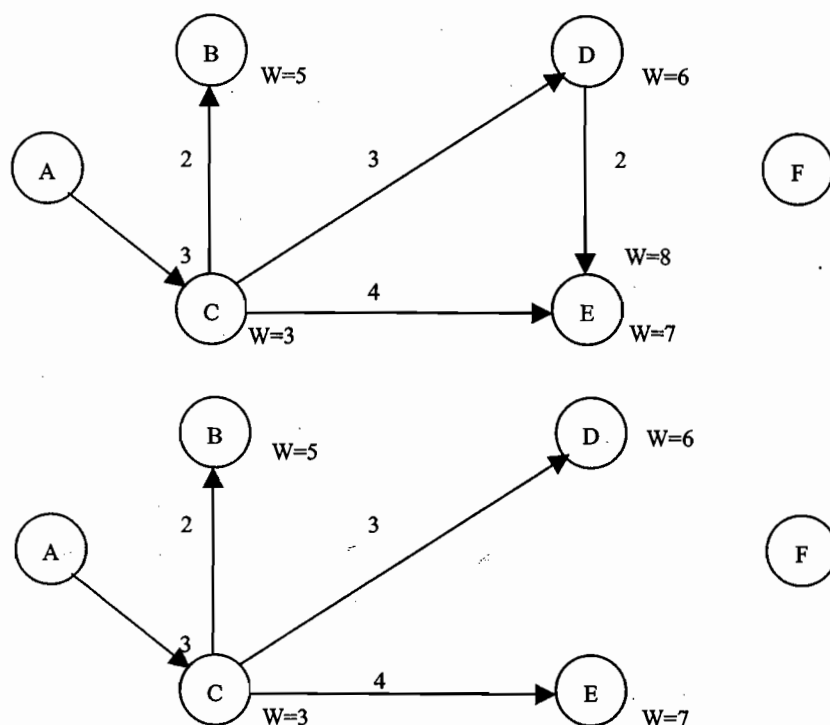
我们再从总和路径最短的城市 C 出发。因为总和路径最短才有可能经过城市到达其它城市的路径还是最短的。我们找出与城市 C 邻近的路径，从图中我们知道从城市 C 到城市 B 的路径为 2，城市 C 到城市 D 的路径为 3，城市 C 到城市 E 的路径为 4。此时我们发现如果从城市 A 经城市 C 到达城市 B 的总和路径为 5，比从城市 A 直接到城市 B 的路径 6 还短，所以我们决定选择从城市 A 经城市 C 到达城市 B。



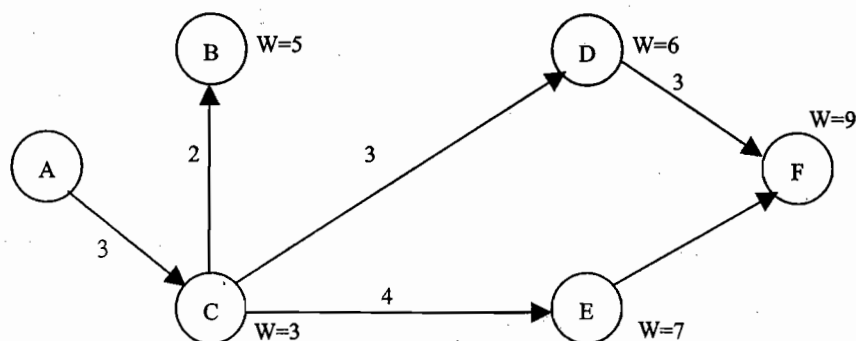
接下来，我们找出与城市 B、城市 C 邻近的路径，从图中我们知道从城市 B 到城市 D 的路径为 5，城市 C 到城市 D 的路径为 3，城市 C 到城市 E 的路径为 4。此时我们发现如果从城市 C 经城市 B 到达城市 D 的总路径为 10，比从城市 C 到达城市 D 的总路径 6 还长，所以我们决定选择从城市 C 直接到达城市 D。

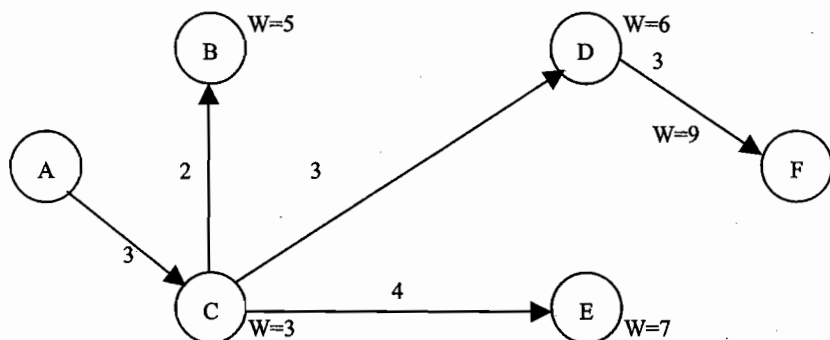


接下来，我们找出与城市 C、城市 D 邻近的路径，从图中我们知道从城市 C 到城市 E 的路径为 4，城市 D 到城市 E 的路径为 2，城市 D 到城市 F 的路径为 3。此时我们发现如果从城市 C 到达城市 E 的总和路径为 7，比从城市 D 到达城市 E 的总和路径 8 还短，所以我们决定选择从城市 C 直接到达城市 E。



接下来，我们找出与城市 D、城市 E 邻近的路径，从图中我们知道从城市 D 到城市 F 的路径为 3，城市 D 到城市 E 的路径为 5。此时我们发现如果从城市 D 到达城市 F 的总和路径为 9，比从城市 E 到达城市 F 的总和路径 12 还短，所以我们决定选择从城市 D 直接到达城市 F。





从上面的运作过程中，我们可以找出从城市 A 到城市 B 的最短路径为 5、从城市 A 到城市 C 的最短路径为 3、从城市 A 到城市 D 的最短路径为 6、从城市 A 到城市 E 的最短路径为 7、从城市 A 到城市 F 的最短路径为 9。

综合上面的过程，我们可以发现这和用 Prim 算法找出最小生成树的作法类似。Dijkstra 算法的规则为：

1. 将起始顶点插入树中。
2. 找出树中所有顶点的邻接边，总和最小的边。
3. 重复第二步骤，直到所有的顶点都在树中为止。

如果我们想要把 Dijkstra 算法写成程序的话，我们可以运用加权边图形的邻接数组来进行找出最短路径的工作。

程序实例：

设计一个利用 Dijkstra 算法找出最短路径的程序。

程序构思：

首先，我们先把图形建立成加权边图形的邻接数组。如下：

|   | 1        | 2        | 3        | 4        | 5        | 6        |
|---|----------|----------|----------|----------|----------|----------|
| 1 | 0        | 6        | 3        | $\infty$ | $\infty$ | $\infty$ |
| 2 | $\infty$ | 0        | $\infty$ | 5        | $\infty$ | $\infty$ |
| 3 | $\infty$ | 2        | 0        | 3        | 4        | $\infty$ |
| 4 | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 3        |
| 5 | $\infty$ | $\infty$ | $\infty$ | 2        | 0        | 5        |
| 6 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

在程序，我们可以将  $\infty$  设为一个极大值(如：999)。

声明一个一维数组来记录已查找的数组(Visited)和一个一维数组用来记录顶点间距离总和的变化(Distance)。

取原先的距离总和与新加入顶点后的距离总和的最小值作为新的距离总和。如本题，如果从顶点 1 出发，则 Distance 的变化如下：

|      | 1 | 2 | 3 | 4        | 5        | 6        |
|------|---|---|---|----------|----------|----------|
| 步骤 1 | 0 | 6 | 3 | $\infty$ | $\infty$ | $\infty$ |
| 步骤 2 | 0 | 5 | 3 | 6        | 7        | $\infty$ |
| 步骤 3 | 0 | 5 | 3 | 6        | 7        | $\infty$ |
| 步骤 4 | 0 | 5 | 3 | 6        | 7        | $\infty$ |
| 步骤 5 | 0 | 5 | 3 | 6        | 7        | 9        |

程序源代码:

```

01 /* ===== Program Description ===== */
02 /* 程序名称: dijkstra.c */
03 /* 程序目的: 设计一个利用 Dijkstra 算法找出最短路径的 */
04 /* 程序。 */
05 /* Written By Kuo-Yu Huang. (WANT Studio.) */
06 /* ===== */
07 #define Max 999 /* 定义最大数 */
08 #define VertexNum 7 /* 定义顶点数 */
09 #define EdgeNum 9 /* 定义邻接边数 */
10
11 int Graph[VertexNum][VertexNum]; /* 图形邻接数组 */
12 int Edge[EdgeNum][3] = /* 输入数据 */
13 { {1,2,6}, {1,3,3}, {2,4,5},
14 {3,2,2}, {3,4,3}, {3,5,4},
15 {4,6,3}, {5,4,2}, {5,6,5} };
16 int Visited[VertexNum]; /* 查找记录 */
17 int Distance[VertexNum]; /* 距离总和 */
18 /* ----- */
19 /* Dijkstra 算法 */
20 /* ----- */
21 void Dijkstra(int Begin)
22 {
23 int MinEdge; /* 最小边 */
24 int Vertex; /* 最小边的顶点 */
25 int i,j;
26 int Edges; /* 边数 */
27
28 Edges = 1; /* 初始边数 */
29 Visited[Begin] = 1; /* 初始顶点 */
30
31 for (i=1;i<VertexNum;i++)
32 Distance[i] = Graph[Begin][i]; /* 初始距离总和 */
33
34 Distance[Begin] = 0; /* 起始点的距离为 0 */
35 printf("Vertex");
36 for (i=1;i<VertexNum;i++)
37 printf("%5d",i); /* 输出顶点数据 */
38 printf("\n");
39 printf("Step %d :",Edges);
40 for (i=1;i<VertexNum;i++)
41 printf("%5d",Distance[i]); /* 输出距离总和和数据 */
42 printf("\n");
43 while (Edges < (VertexNum - 1)) /* 当边数少于顶点数时执行 */
44 {
45 Edges++;
46 MinEdge = Max; /* 将最小边设到最大值 */
47 for (j=1;j<VertexNum;j++) /* 判断未建立的邻接顶点 */

```

```

49 {
50 /* 顶点未查找过且最小边加权值比距离总和大 */
51 if (Visited[j] == 0 && MinEdge > Distance[j])
52 {
53 Vertex = j; /* 找出最小边的顶点 */
54 MinEdge = Distance[j]; /* 找出最小边的距离总和 */
55 }
56 }
57 Visited[Vertex] = 1; /* 将最小边的顶点设为已查找 */
58 printf("Step %d :",Edges);
59 for (j=1;j<VertexNum;j++)
60 {
61 /* 找出未查找顶点的最小距离总和 */
62 if (Visited[j] == 0 &&
63 Distance[Vertex] + Graph[Vertex][j] < Distance[j])
64 {
65 Distance[j] =
66 Distance[Vertex] + Graph[Vertex][j];
67 }
68 printf("%5d",Distance[j]);
69 }
70 printf("\n");
71 }
72 }
73 }
74
75 /* ----- */
76 /* 输出邻接数组数据 */
77 /* ----- */
78 void Print_M_Graph()
79 {
80 int i,j;
81
82 printf("Vertice");
83 for (i=1;i<VertexNum;i++)
84 printf("%5d",i);
85 printf("\n");
86 for (i=1;i<VertexNum;i++)
87 {
88 printf("%5d ",i);
89 for (j=1;j<VertexNum;j++)
90 printf("%5d",Graph[i][j]);
91 printf("\n");
92 }
93 }
94
95 /* ----- */
96 /* 以邻接数组建立图形 */
97 /* ----- */
98 void Create_M_Graph(int Vertice1,int Vertice2,int Weight)
99 {
100
101 Graph[Vertice1][Vertice2] = Weight; /* 数组内容 */
102 }
103
104 /* ----- */
105 /* 主程序 */
106 /* ----- */
107 void main ()
108 {
109 int BeginVertex = 1; /* 起始顶点 */

```



```

110 int i,j;
111
112 for (i=0;i<VertexNum;i++) /* 清除查找记录 */
113 Visited[i] = 0;
114
115 for (i=0;i<VertexNum;i++) /* 清除数组数据 */
116 for (j=0;j<VertexNum;j++)
117 Graph[i][j] = Max;
118
119 for (i=0;i<EdgeNum;i++) /* 调用建立邻接数组 */
120 Create_M_Graph(Edge[i][0],Edge[i][1],Edge[i][2]);
121
122 printf("##Graph##\n");
123 Print_M_Graph(); /* 调用输出邻接数组数据 */
124
125 printf("Dijkstra Algorithm : \n");
126 Dijkstra(BeginVertex); /* 调用 Dijkstra */
127 }

```

运行结果:

```

C:\DS>dijkstra
##Graph##
Vertice 1 2 3 4 5 6
1 999 6 3 999 999 999
2 999 999 999 5 999 999
3 999 2 999 3 4 999
4 999 999 999 999 999 3
5 999 999 999 2 999 5
6 999 999 999 999 999 999

Dijkstra Algorithm :
Vertice 1 2 3 4 5 6
Step 1 : 0 6 3 999 999 999
Step 2 : 0 5 3 6 7 999
Step 3 : 0 5 3 6 7 999
Step 4 : 0 5 3 6 7 9
Step 5 : 0 5 3 6 7 9
Step 6 : 0 5 3 6 7 9

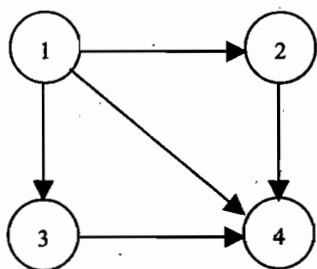
C:\DS>

```

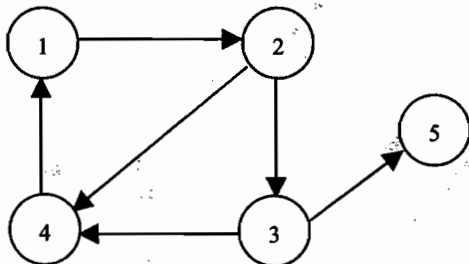
## 【习题】

一、复习:

1. 图形是由顶点和边(Edges 或 Arcs)所构成的两个有限非空集合。
2. 下面图形的集合表示法为:  
 $V(G) = \{1, 2, 3, 4\}$   
 $E(G) = \{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)\}$



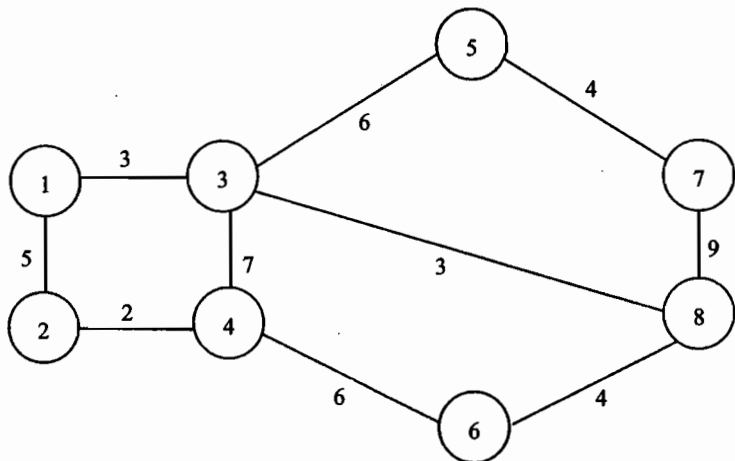
3. 对于一个有  $N$  个顶点的无向完全图形，其边数为  $N(N-1)/2$ 。
4. 对于一个具有  $M$  个顶点和  $N$  个邻接边的无向图形，若用邻接列表表示法表示，则需要  $M$  个 Head 节点和  $2N$  个列表节点。
5. 具有加权边的图形，如果用邻接数组表示法，数组中的元素即为加权值；如果用邻接列表表示法，则列表节点需增加一个加权值字段。
6. 使用深度优先搜索法是用队列来存储未查找的邻接顶点。
7. 关于下列图形，哪一个正确？



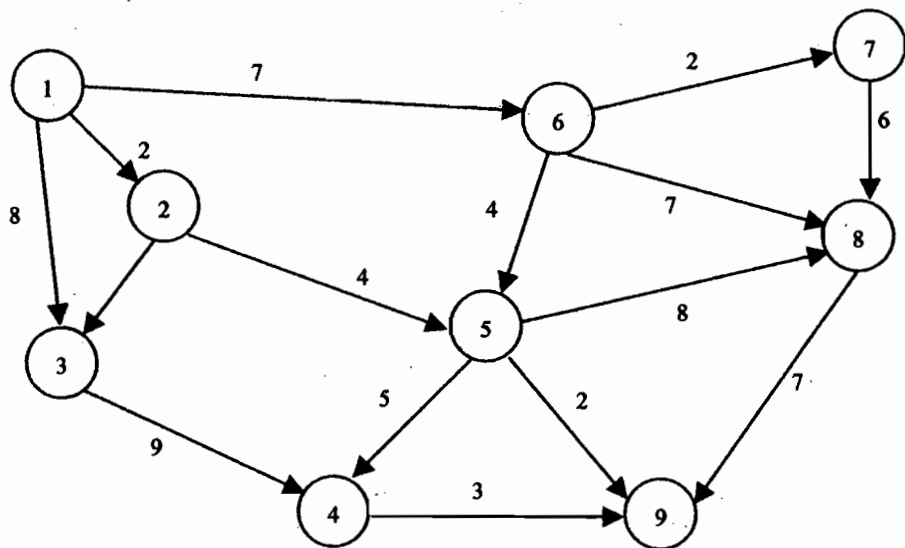
- (a) 路径  $\langle V_1, V_2 \rangle, \langle V_2, V_4 \rangle, \langle V_4, V_1 \rangle$  是一条回路。
- (b) 顶点 2 的内分支度为 2。
- (c) 顶点 3 的外分支度为 2。
- (d) 以上皆非。

## 二、应用：

有一个无向图形如下(试回答下列题目 1~5)：

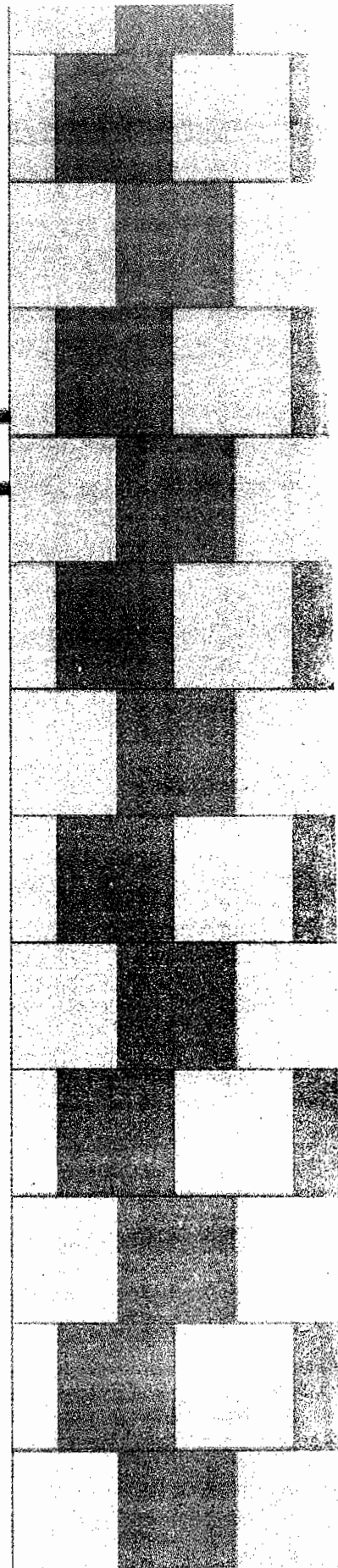


1. 试画出深度优先搜索的图形。(从顶点 1)
2. 试画出广度优先搜索的图形。(从顶点 1)
3. 试画出深度优先搜索的图形。(从顶点 8)
4. 试画出广度优先搜索的图形。(从顶点 8)
5. 试画出最小生成树的图形。
6. 试找出下图顶点 1 到各顶点间的最短路径。



ASCII 码

附 录 A



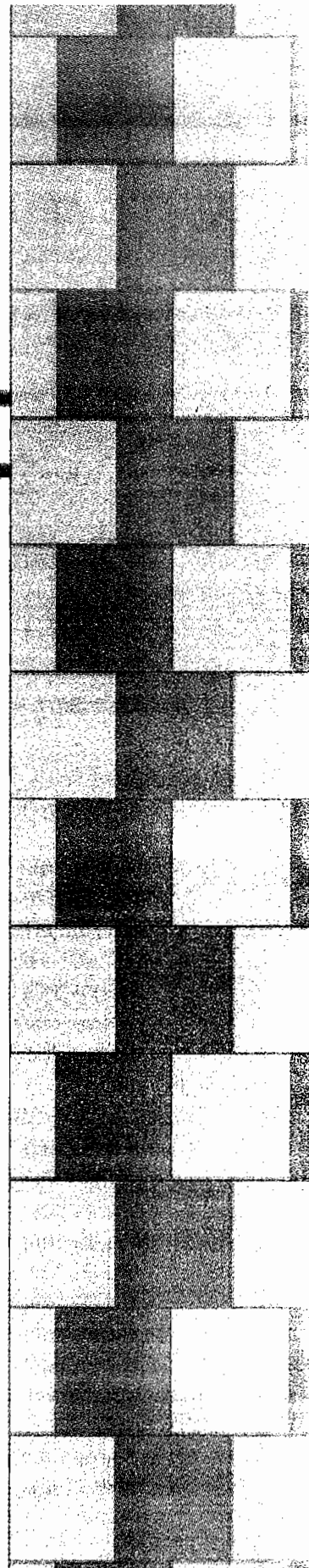
| 10 进制 | 8 进制 | 16 进制 | ASCII | 10 进制 | 8 进制 | 16 进制 | ASCII |
|-------|------|-------|-------|-------|------|-------|-------|
| 0     | 0    | 0     |       | 54    | 66   | 36    | 6     |
| 1     | 1    | 1     | ☺     | 55    | 67   | 37    | 7     |
| 2     | 2    | 2     | ☺     | 56    | 70   | 38    | 8     |
| 3     | 3    | 3     | ♥     | 57    | 71   | 39    | 9     |
| 4     | 4    | 4     | ♦     | 58    | 72   | 3a    | :     |
| 5     | 5    | 5     | ♣     | 59    | 73   | 3b    | ;     |
| 6     | 6    | 6     | ♣     | 60    | 74   | 3c    | <     |
| 7     | 7    | 7     | ●     | 61    | 75   | 3d    | =     |
| 8     | 10   | 8     | ■     | 62    | 76   | 3e    | >     |
| 9     | 11   | 9     | ○     | 63    | 77   | 3f    | ?     |
| 10    | 12   | a     | ☐     | 64    | 100  | 40    | @     |
| 11    | 13   | b     | ♂     | 65    | 101  | 41    | A     |
| 12    | 14   | c     | ♀     | 66    | 102  | 42    | B     |
| 13    | 15   | d     | ♂     | 67    | 103  | 43    | C     |
| 14    | 16   | e     | ♂     | 68    | 104  | 44    | D     |
| 15    | 17   | f     | ☼     | 69    | 105  | 45    | E     |
| 16    | 20   | 10    | ▶     | 70    | 106  | 46    | F     |
| 17    | 21   | 11    | ◀     | 71    | 107  | 47    | G     |
| 18    | 22   | 12    | ↑↓    | 72    | 110  | 48    | H     |
| 19    | 23   | 13    | !!    | 73    | 111  | 49    | I     |
| 20    | 24   | 14    | ¶     | 74    | 112  | 4a    | J     |
| 21    | 25   | 15    | §     | 75    | 113  | 4b    | K     |
| 22    | 26   | 16    | -     | 76    | 114  | 4c    | L     |
| 23    | 27   | 17    | ↑↓    | 77    | 115  | 4d    | M     |
| 24    | 30   | 18    | ↑     | 78    | 116  | 4e    | N     |
| 25    | 31   | 19    | ↓     | 79    | 117  | 4f    | O     |
| 26    | 32   | 1a    | →     | 80    | 120  | 50    | P     |
| 27    | 33   | 1b    | ←     | 81    | 121  | 51    | Q     |
| 28    | 34   | 1c    | ↖     | 82    | 122  | 52    | R     |
| 29    | 35   | 1d    | ↗     | 83    | 123  | 53    | S     |
| 30    | 36   | 1e    | ▲     | 84    | 124  | 54    | T     |
| 31    | 37   | 1f    | ▼     | 85    | 125  | 55    | U     |
| 32    | 40   | 20    | .     | 86    | 126  | 56    | V     |
| 33    | 41   | 21    | !     | 87    | 127  | 57    | W     |
| 34    | 42   | 22    | "     | 88    | 130  | 58    | X     |
| 35    | 43   | 23    | #     | 89    | 131  | 59    | Y     |
| 36    | 44   | 24    | \$    | 90    | 132  | 5a    | Z     |
| 37    | 45   | 25    | %     | 91    | 133  | 5b    | [     |
| 38    | 46   | 26    | &     | 92    | 134  | 5c    | \     |
| 39    | 47   | 27    | '     | 93    | 135  | 5d    | ]     |
| 40    | 50   | 28    | (     | 94    | 136  | 5e    | ^     |
| 41    | 51   | 29    | )     | 95    | 137  | 5f    | ~     |
| 42    | 52   | 2a    | *     | 96    | 140  | 60    |       |
| 43    | 53   | 2b    | +     | 97    | 141  | 61    | a     |
| 44    | 54   | 2c    | ,     | 98    | 142  | 62    | b     |
| 45    | 55   | 2d    | -     | 99    | 143  | 63    | c     |
| 46    | 56   | 2e    | .     | 100   | 144  | 64    | d     |
| 47    | 57   | 2f    | /     | 101   | 145  | 65    | e     |
| 48    | 60   | 30    | 0     | 102   | 146  | 66    | f     |
| 49    | 61   | 31    | 1     | 103   | 147  | 67    | g     |
| 50    | 62   | 32    | 2     | 104   | 150  | 68    | h     |
| 51    | 63   | 33    | 3     | 105   | 151  | 69    | i     |
| 52    | 64   | 34    | 4     | 106   | 152  | 6a    | j     |
| 53    | 65   | 35    | 5     | 107   | 153  | 6b    | k     |

| 10 进制 | 8 进制 | 16 进制 | ASCII | 10 进制 | 8 进制 | 16 进制 | ASCII |
|-------|------|-------|-------|-------|------|-------|-------|
| 108   | 154  | 6c    | l     | 162   | 242  | a2    | ó     |
| 109   | 155  | 6d    | m     | 163   | 243  | a3    | ú     |
| 110   | 156  | 6e    | n     | 164   | 244  | a4    | ñ     |
| 111   | 157  | 6f    | o     | 165   | 245  | a5    | Ñ     |
| 112   | 160  | 70    | p     | 166   | 246  | a6    | ª     |
| 113   | 161  | 71    | q     | 167   | 247  | a7    | º     |
| 114   | 162  | 72    | r     | 168   | 250  | a8    | ¿     |
| 115   | 163  | 73    | s     | 169   | 251  | a9    | ¸     |
| 116   | 164  | 74    | t     | 170   | 252  | aa    | ¸     |
| 117   | 165  | 75    | u     | 171   | 253  | ab    | ½     |
| 118   | 166  | 76    | v     | 172   | 254  | ac    | ¼     |
| 119   | 167  | 77    | w     | 173   | 255  | ad    | ¡     |
| 120   | 170  | 78    | x     | 174   | 256  | ae    | «     |
| 121   | 171  | 79    | y     | 175   | 257  | af    | »     |
| 122   | 172  | 7a    | z     | 176   | 260  | b0    | ■     |
| 123   | 173  | 7b    | {     | 177   | 261  | b1    | ■     |
| 124   | 174  | 7c    |       | 178   | 262  | b2    | ■     |
| 125   | 175  | 7d    | }     | 179   | 263  | b3    | ■     |
| 126   | 176  | 7e    | ~     | 180   | 264  | b4    | ┌     |
| 127   | 177  | 7f    | Δ     | 181   | 265  | b5    | ┌     |
| 128   | 200  | 80    | Ç     | 182   | 266  | b6    | ┌     |
| 129   | 201  | 81    | ü     | 183   | 267  | b7    | ┌     |
| 130   | 202  | 82    | é     | 184   | 270  | b8    | ┌     |
| 131   | 203  | 83    | â     | 185   | 271  | b9    | ┌     |
| 132   | 204  | 84    | ä     | 186   | 272  | ba    | ┌     |
| 133   | 205  | 85    | à     | 187   | 273  | bb    | ┌     |
| 134   | 206  | 86    | å     | 188   | 274  | bc    | ┌     |
| 135   | 207  | 87    | ç     | 189   | 275  | bd    | ┌     |
| 136   | 210  | 88    | ê     | 190   | 276  | be    | ┌     |
| 137   | 211  | 89    | ë     | 191   | 277  | bf    | ┌     |
| 138   | 212  | 8a    | è     | 192   | 300  | c0    | ┌     |
| 139   | 213  | 8b    | ï     | 193   | 301  | c1    | ┌     |
| 140   | 214  | 8c    | î     | 194   | 302  | c2    | ┌     |
| 141   | 215  | 8d    | ì     | 195   | 303  | c3    | ┌     |
| 142   | 216  | 8e    | Ä     | 196   | 304  | c4    | ┌     |
| 143   | 217  | 8f    | Å     | 197   | 305  | c5    | ┌     |
| 144   | 220  | 90    | É     | 198   | 306  | c6    | ┌     |
| 145   | 221  | 91    | æ     | 199   | 307  | c7    | ┌     |
| 146   | 222  | 92    | Æ     | 200   | 310  | c8    | ┌     |
| 147   | 223  | 93    | ô     | 201   | 311  | c9    | ┌     |
| 148   | 224  | 94    | ö     | 202   | 312  | ca    | ┌     |
| 149   | 225  | 95    | ò     | 203   | 313  | cb    | ┌     |
| 150   | 226  | 96    | û     | 204   | 314  | cc    | ┌     |
| 151   | 227  | 97    | ù     | 205   | 315  | cd    | ┌     |
| 152   | 230  | 98    | ÿ     | 206   | 316  | ce    | ┌     |
| 153   | 231  | 99    | Ö     | 207   | 317  | cf    | ┌     |
| 154   | 232  | 9a    | Ü     | 208   | 320  | d0    | ┌     |
| 155   | 233  | 9b    | ç     | 209   | 321  | d1    | ┌     |
| 156   | 234  | 9c    | £     | 210   | 322  | d2    | ┌     |
| 157   | 235  | 9d    | ¥     | 211   | 323  | d3    | ┌     |
| 158   | 236  | 9e    | ₣     | 212   | 324  | d4    | ┌     |
| 159   | 237  | 9f    | ₣     | 213   | 325  | d5    | ┌     |
| 160   | 240  | a0    | á     | 214   | 326  | d6    | ┌     |
| 161   | 241  | a1    | í     | 215   | 327  | d7    | ┌     |

| 10 进制 | 8 进制 | 16 进制 | ASCII | 10 进制 | 8 进制 | 16 进制 | ASCII   |
|-------|------|-------|-------|-------|------|-------|---------|
| 216   | 330  | d8    | ⦿     | 236   | 354  | ec    | ∞       |
| 217   | 331  | d9    | ┐     | 237   | 355  | ed    | φ       |
| 218   | 332  | da    | ┌     | 238   | 356  | ee    | ε       |
| 219   | 333  | db    | ■     | 239   | 357  | ef    | ∩       |
| 220   | 334  | dc    | ■     | 240   | 360  | f0    | ≡       |
| 221   | 335  | dd    | ■     | 241   | 361  | f1    | ±       |
| 222   | 336  | de    | ■     | 242   | 362  | f2    | ≥       |
| 223   | 337  | df    | ■     | 243   | 363  | f3    | ≤       |
| 224   | 340  | e0    | α     | 244   | 364  | f4    | ┐       |
| 225   | 341  | e1    | β     | 245   | 365  | f5    | ┐       |
| 226   | 342  | e2    | Γ     | 246   | 366  | f6    | ÷       |
| 227   | 343  | e3    | π     | 247   | 367  | f7    | ∞       |
| 228   | 344  | e4    | Σ     | 248   | 370  | f8    | °       |
| 229   | 345  | e5    | σ     | 249   | 371  | f9    | ·       |
| 230   | 346  | e6    | μ     | 250   | 372  | fa    | ·       |
| 231   | 347  | e7    | τ     | 251   | 373  | fb    | ·       |
| 232   | 350  | e8    | Φ     | 252   | 374  | fc    | η       |
| 233   | 351  | e9    | Θ     | 253   | 375  | fd    | ²       |
| 234   | 352  | ea    | Ω     | 254   | 376  | fe    | ■       |
| 235   | 353  | eb    | δ     | 255   | 377  | ff    | (blank) |

# 习 题 解 答

## 附 录 B





## 第1章 数据结构的基本概念习题解答

一、复习(是非、选择):

1. 对。
2. 错。伪码(Pseudo Code)是以夹杂程序语法和自然语言(如: 中文、英文)来描述解决问题的方法。
3. 对。共执行  $n$  次, 所以时间复杂度为  $O(n)$ 。

$$\begin{aligned}
 4. \quad & \text{对。共执行 } \sum_{i=1}^n \sum_{j=i}^n \sum_{k=j}^n 1 = \sum_{i=1}^n \sum_{j=i}^n (n-j+1) = \sum_{i=1}^n \left( \sum_{j=i}^n (-j) + \sum_{j=i}^n (n+1) \right) \\
 &= \sum_{i=1}^n \left( -\frac{(i+n)(n-i+1)}{2} + (n+1)(n-i+1) \right) \\
 &= \square \sum_{i=1}^n i^2 + \square \sum_{i=1}^n i + \square \sum_{i=1}^n 1
 \end{aligned}$$

所以只要考虑最大项  $\sum_{i=1}^n i^2$ ,  $\sum_{i=1}^n i^2 = \frac{n(n+1)(n+2)}{6} \in O(n^3)$

5. 错。如果试着去执行这个程序, 会发觉这个程序是一个无穷循环。
6. 执行次数为  $\sum_{i=1}^n \log_2 n \in O(\log_2 n)$ 。
7. (a)(b)(c)。输入(Input)、输出(Output)、定义明确(Definiteness)、有限的步骤(Finiteness)、有效率的步骤(Effectiveness), 5个步骤。
8. (a)(c)(d)。题目所问的是影响“单一指令”执行的时间, 所以(b)循环范围不符合。
9. (d)。顺序查找法中最佳状况的时间复杂度(Best-case time complexity)为  $B(n) = 1 \in O(1)$ 、最坏状况的时间复杂度(Worst-case time complexity)为  $W(n) = n \in O(n)$ 、平均状况的时间复杂度(Average-case time complexity)为  $A(n) = (n+1)/2 \in O(n)$ 。但一般状况的时间复杂度(Every-case time complexity)不存在。
10. (d)。

这就是由大至小的排序程序, 执行过程如下:

原数据:  $X[0] = 5, X[1] = 9, X[2] = 2, X[3] = 6, X[4] = 7, X[5] = 4$

第1步:  $X[0] = 9, X[1] = 5, X[2] = 2, X[3] = 6, X[4] = 7, X[5] = 4$

第2步:  $X[0] = 9, X[1] = 6, X[2] = 2, X[3] = 5, X[4] = 7, X[5] = 4$

第3步:  $X[0] = 9, X[1] = 7, X[2] = 2, X[3] = 5, X[4] = 6, X[5] = 4$

第4步:  $X[0] = 9, X[1] = 7, X[2] = 5, X[3] = 2, X[4] = 6, X[5] = 4$

第5步:  $X[0] = 9, X[1] = 7, X[2] = 6, X[3] = 2, X[4] = 5, X[5] = 4$

第6步:  $X[0] = 9, X[1] = 7, X[2] = 6, X[3] = 5, X[4] = 2, X[5] = 4$

第7步:  $X[0] = 9, X[1] = 7, X[2] = 6, X[3] = 5, X[4] = 4, X[5] = 2$

11. 使用找第  $k$  小元素的算法平均为  $O(n)$ 。中位数即取  $k=n/2$ 。

12. (c)。

因为  $a^n < n! < n^n$ , 所以  $n! \in O(a^n)$  不符合。

因为  $n^{0.00001} > \log n$ , 所以  $n^{0.00001} \in O(\log n)$  不符合。

$$1 + 1/2 + 1/3 + 1/4 + \cdots + 1/n = \sum_{i=1}^n \frac{1}{i} \in O(\log n).$$

因为  $3^n > 2^n$ , 所以  $3^n \in O(2^n)$ 。

## 第 2 章 数组习题解答

### 一、复习:

1. 对。数组的使用是一种静态的内存空间配置。
2. 错。数组是存储同一类型数据的数据结构。
3. 错。不一定要全都存放数据。
4. 对。数组的表示法, 有以行为主(Column-Major)和以列为主(Row-Major)两种。
5. 错。以行为主的数组表示法比以列为主的数组表示法所用的空间相等。
6. 错。其可使用的空间为 Data[0]~Data[19]。
7. (c)。内容值不是该笔被存储数据的位置, 而是该笔被存储的数据内容。
8. (b)
9. (b)。依题意知, 数组 X 是一个整数数组, 则每个元素所占的位置为 2bytes。

已知 X[3,5]的地址为 1000, X[5,7]的地址为 1200

$X[i][j]$  = 数组第一个元素位置 + [(i \* 每一列的元素个数) + j] \* (所声明数据类型所占的大小)  
将已知代入得:

$$X[3,5] = 1000 = M + [(3*n)+5]*2 = M + 6*n + 10 \cdots \cdots (i)$$

$$X[5,7] = 1200 = M + [(5*n)+7]*2 = M + 10*n + 14 \cdots \cdots (ii)$$

解(i)和(ii)得: 数组第一个元素位置 M = 706、每一列的元素个数 n = 49。

$$X[8,10] = 706 + (8 * 49 + 5) * 2 = 1500$$

$$X[3,8] = 706 + (3 * 49 + 8) * 2 = 1016$$

## 第 3 章 链表习题解答

### 一、复习:

1. 对。
2. (b) NEW->Next = Pointer->Next  
Pointer->Next = NEW
3. (d) Back->Next = Pointer->Next;  
free(Pointer);
4. (c)(d)

## 第4章 堆栈习题解答

### 一、复习:

#### 1. 定义:

“堆栈”是在程序设计时经常使用的数据结构，为一有序列表，堆栈数据结构的特性是只允许数据自有序列表之一固定端(前端)做输入、输出动作，如此一来会使得最后被输入的数据项会最先被取出来，也就是具有 LIFO 的特性。

堆栈的应用:

##### (1) 子程序之调用:

在跳往子程序前，会先将下一个指令的地址存到堆栈中，直到子程序执行完后再将地址取出，以回到原来的程序中。

##### (2) 处理递归调用:

和子程序之调用类似，只是除了存储下一个指令的地址外，也将参数、区域变量等数据存入堆栈中。

##### (3) 表达式之转换与求值

##### (4) 二叉树的遍历

##### (5) 图形之深度优先追踪法

#### 2. 数组，链表

#### 3. FILO(First In Last Out)

#### 4. 1, 2, 3, 4, 5

## 第5章 队列习题解答

### 一、复习:

#### 1. 定义:

队列数据结构规定在有序列表中数据之输出、输入是分别由不同端进行处理，输出端称为前端(front)，输入端称为后端(rear)，这样的状况会使得先存入的数据会先被取出，也就是具有先进先出 FIFO 的特性。

队列的应用:

图形之宽度优先追踪法。

优先队列，此种队列在取出元素时是根据所存元素的某项特性值或优先权而取出具最小或最大数值的元素。

操作系统中之工作调度，若工作之优先权相同，则采用先到先做的原则。

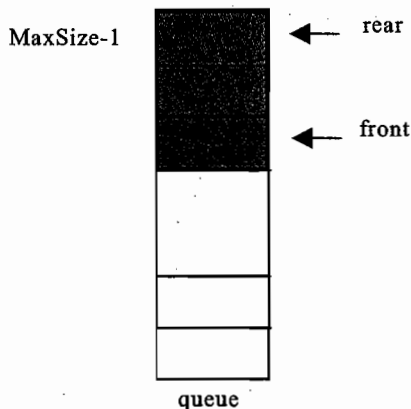
用于“spooling”，先将输出数据写在磁盘上，再由打印机以先存入者先处理的顺利将数据输出。

#### 2. 数组，链表

#### 3. FIFO

#### 4. 5, 4, 3, 2, 1

5. 当队尾指针  $rear$  等于  $MaxSize$  时, 不论队列是否仍有空间, 均无法再将数据存于队列中, 如下图:



如果要将数据往队列前端移动以挪出空间, 需要花费很多时间。为解决这样的问题, 我们使用环状队列, 控制队列前尾指针  $front$ 、 $rear$  来充份运用队列中的空间。

## 第6章 递归习题解答

### 一、复习:

1. 对。
2. 对。递归程序所使用的是一种反复的调用子程序的结构, 所以需要运用堆栈来记录程序的返回地址。
3. 错。非递归程序可能仅需要用单一循环就可以处理, 而且不需要用堆栈记录返回地址, 所以时间复杂度和空间复杂度比递归程序有效率。
4. (b)。当  $n = 0$  时, 执行一次, 当  $n$  大于 0 时, 则执行  $n+1$  次。 $fact(8)=40320$ , 因为函数声明为整数, 所以最大可计算数值不可超过 32767。
5. (d)。  
调用次数  $T(n)$ , 当  $n \leq 3$  时, 执行一次。当  $n > 3$  时, 则递归调用  $T(n-2) + T(n-4) + 1$ 。

| N       | $T(n)$ |               |
|---------|--------|---------------|
| 0,1,2,3 | 1      | 1             |
| 4       | 3      | $T(2)+T(0)+1$ |
| 5       | 3      | $T(3)+T(1)+1$ |
| 6       | 5      | $T(4)+T(2)+1$ |
| 7       | 5      | $T(5)+T(3)+1$ |
| 8       | 9      | $T(6)+T(4)+1$ |
| 9       | 9      | $T(7)+T(5)+1$ |

$X(X(8))$  中的  $X(8)=9$ , 执行 9 次

$X(9)=9$ , 再执行 9 次。共执行 18 次。

## 6. (b)(c).

第一步为将 1 号铁盘从 A 塔搬至 C 塔。

第二步为将 2 号铁盘从 A 塔搬至 B 塔。

第 3 步为将 1 号铁盘从 C 塔搬至 B 塔。

第 4 步为将 3 号铁盘从 A 塔搬至 C 塔。

第 5 步为将 1 号铁盘从 B 塔搬至 C 塔。

第 6 步为将 2 号铁盘从 B 塔搬至 C 塔。

第 7 步为将 1 号铁盘从 A 塔搬至 C 塔。

需要 7 次才完成工作。

## 7. (b).

$$\text{maze}(1024,10,7) = 7 * \text{maze}(102,10,7) + 0 = 7 * (7 * \text{maze}(10,10,7) + 2) + 0$$

$$= 7 * (7 * (7 * \text{maze}(1,10,7) + 0) + 2) + 0$$

$$= 7 * (7 * (7 * 1 + 2) + 0) + 0 = 7 * (7 * 7 + 2) + 0$$

$$= 7 * 51 = 357$$

$$\text{maze}(352,4,11) = 11 * \text{maze}(88,4,11) = 11 * (11 * \text{maze}(22,4,11))$$

$$= 11 * (11 * (11 * \text{maze}(5,4,11) + 2))$$

$$= 11 * (11 * (11 * (11 * \text{maze}(1,4,11) + 1) + 2))$$

$$= 11 * (11 * (11 * (11 * 1 + 1) + 2))$$

$$= 11 * (11 * (11 * 12 + 2))$$

$$= 11 * (11 * 134) = 16214$$

$$\text{maze}(16,2,2) = 2 * \text{maze}(8,2,2) = 2 * (2 * \text{maze}(4,2,2))$$

$$= 2 * (2 * (2 * \text{maze}(2,2,2)))$$

$$= 2 * (2 * (2 * 2))$$

$$= 2 * (2 * 4) = 2 * 8 = 16$$

## 第 7 章 基础树状结构习题解答

### 一、复习:

1. 错。树不可为空，至少要有有一个根节点(root)
2. 对。
3. 错。二叉树的子树有顺序关系，而一般树没有
4. 对。
5. 错。不一定，完全二叉树为满二叉树之子集，当完全二叉树最大阶层的所有节点均存在时才为满二叉树。
6. (b)
7. (c)
8. (d)

证明:  $n = n_0 + n_1 + n_2$

$$\text{Branch} = n - 1 = n_0 + n_1 + n_2 - 1 \quad \text{-----(1)}$$

$$\text{Branch} = 0 \cdot n_0 + 1 \cdot n_1 + 2 \cdot n_2 \quad \text{-----(2)}$$

$$= n_1 + 2 \cdot n_2$$

由(1)(2)可得:

$$n_0 + n_1 + n_2 - 1 = n_1 + 2 \cdot n_2$$

$$\rightarrow n_0 = n_2 + 1$$

9. (b)

10.

(1) 树(Tree):

树是由一个或多个节点(node)所构成之有限集合。每一棵树必有一特定的节点, 称做根节点(root)。根节点之下可以有零个以上的子节点(可以没有), 而各子节点也可为子树, 拥有自己的子节点。

(2) 元树(Binary tree):

二叉树是树的一种, 其节点至多只能有两个子节点。是由有限个节点所构成之集合, 此集合可以为空的。二叉树的根节点(root)下可分成两个子树, 称为左子树和右子树, 左子树和右子树有顺序关系。

(3) 满二叉树(Full binary tree):

一树中所有叶节点均在同一阶层, 而其它非终端节点(nonterminal node)之分支度(degree)均为2, 则此树为一满二叉树。

(4) 完全二叉树(Complete binary tree):

一树扣除掉最大 level 那层后为一满二叉树, 且 level 最大那层之节点均向左靠齐, 则该二叉树称为完全二叉树。

(5) 歪斜树(Skewed tree):

节点分布不均所造成的结果。例如在一棵树中, 若所有节点之 L 均不存在, 则此树为右歪斜树(right skewed binary tree), 反之, 所有节点之 R 均不存在, 则此树为左歪斜树(left skewed binary tree)。

(6) 引线二叉树(Threaded binary tree):

引线二叉树充份应用链接结构二叉树中的空指针以建立引线, 使二叉树中的节点能有更紧密的连结, 并且能够加快二叉树的遍历速度。

## 第 8 章 排序习题解答

一、复习:

1. 所谓排序是将一群数据, 依指定顺序所进行的排列过程。最常见的有“由小到大”的“递增顺序”和“由大到小”的“递减顺序”。
2. 所谓“稳定性排序”即是排序过后能使值相同的数据, 保持原顺序中之相对位置。反之, 则称为不稳定性排序。

举例说明:

| data | 0  | 1  | 2    | 3  | 4 | 5  | 6    | 7  |
|------|----|----|------|----|---|----|------|----|
|      | 38 | 16 | 7(1) | 66 | 2 | 24 | 7(2) | 19 |

7(1)和 7(2)为值相同的数据

顺序: 7(1)在 7(2)之前

● 稳定性排序的结果:

|      |   |      |      |    |    |    |    |    |
|------|---|------|------|----|----|----|----|----|
|      | 0 | 1    | 2    | 3  | 4  | 5  | 6  | 7  |
| data | 2 | 7(1) | 7(2) | 16 | 19 | 24 | 38 | 66 |

顺序: 7(1)在 7(2)之前 → 维持原来的相对位置

● 不稳定性排序的结果:

|      |   |      |      |    |    |    |    |    |
|------|---|------|------|----|----|----|----|----|
|      | 0 | 1    | 2    | 3  | 4  | 5  | 6  | 7  |
| data | 2 | 7(2) | 7(1) | 16 | 19 | 24 | 38 | 66 |

顺序: 7(2)在 7(1)之前 → 和原来的相对位置不同

3.  $n, n^2 + n$

4. (b)

(a) 外部排序需要抽取外部存储装置的数据, 故处理速度较内部排序慢

(b) 正确

(c) 两者处理机制相同, 均为进行数据的排序

(d) 外部存储装置不可以随机存取数据

5. 排序的分类大致上可分为两种:

(1) 内部排序 (Internal Sort)

将欲处理的数据整个存放到内部存储器中做排序, 数据可被随机存取。

例如:

(a) 交换式排序法

(b) 选择式排序法

(c) 插入式排序法

(2) 外部排序 (External Sort)

欲处理的数据量过于庞大, 无法全部存放到内部存储器, 必须借助外部辅助内存 (ex: 磁盘, 磁带), 数据不可随机被存取。

例如:

(a) 合并排序法

(b) 直接合并排序法

## 第 9 章 查找习题解答

一、复习:

1. 错。线性查找法需考虑数据是否已经排序好, 才可以进行数据查找工作。
2. 对。
3. 错。折半查找法的最坏状态的时间复杂度为  $W(n) = n \in O(\log n)$ 。
4. 错。  $4 < \log_2 30 < 5$ , 5 次
5. 对。

6. 错。插补查找法对于查找平均分布的数据较有效率。
7. 错。在没有杂凑碰撞时，最少只需一次。
8. (b)(c)(d)。
9. (b)  
 $(5 + 1^2) \% 11 = (5 + 1) \% 11 = 6$   
 $(6 + 2^2) \% 11 = (6 + 4) \% 11 = 10$   
 $(10 + 3^2) \% 11 = (10 + 9) \% 11 = 8$

## 第 10 章 高级链表习题解答

一、复习：

1. (b) 2. (a) 3. (a) 4. (c) 5. (b) 6. (c)

## 第 11 章 字符串结构习题解答

一、复习：

1. "\0"
2. 传址
3. string.h
4. scanf(), gets()
5. printf(), puts()
6. 错。字符串是由一连串的字符集
7. 对。字符串的表示方式有“数组结构”和“指针”两种。
8. 错。strlen() 计算出的字符串长度不包含结束字符“\0”。
9. (a)错 (b)对 (c)对 (d)对 (e)错  
 (a) 没有设置数组长度，则需要设置字符串初值  
 比如：char string[]="happy";  
 (e) 若是声明指针，则要去掉中括号“[]”。  
 若是声明字符串，则要去掉星号“\*”，且要设置数组长度或是字符串初值
10. (a)错 (2)对 (3)对 (4)对 (5)错  
 字符串输入的格式有两种：  
 scanf("%s", 字符串变量);  
 gets(字符串变量);
11. (a)对 (b)对 (c)错 (d)错  
 r 是指针，可直接指到字符串变量名称，故(1)(2)是正确的，而 s 是已设置长度的字符串数组，无法直接指到字符串变量名称。



12.

|     | 字节空间 | 字符串长度 |
|-----|------|-------|
| (a) | 10   | 5     |
| (b) | 80   | 10    |
| (c) | 5    | 5     |
| (d) | 8    | 8     |
| (e) | 5    | 0     |

## 第 12 章 图形结构习题解答

一、复习:

- 对。图形的定义为在图形  $G$  中包含了两个集合，一个是由顶点(Vertexes 或 nodes)所构成的有限的非空集合，另一个是由边(Edges 或 Arcs)所构成的有限非空集合。
- 错。这是一张有向图所以集合表示法为：  
 $V(G) = \{1, 2, 3, 4\}$   
 $E(G) = \{ \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 4 \rangle, \langle 3, 4 \rangle \}$
- 对。无向完全图形，边数  $= (N-1) + (N-2) + \cdots + 1 = N(N-1)/2$ 。
- 对。参考本章邻接列表表示法。
- 对。
- 错。设计深度优先搜索程序时，我们可采用堆栈来存储未查找的邻接顶点，或者采用递归来调用深度优先搜索函数，查找未曾查找过的顶点。
- (a)(c)。  
 (a) 路径  $\langle V1, V2 \rangle, \langle V2, V4 \rangle, \langle V4, V1 \rangle$ ，因为起点和终点相同，所以是回路。  
 (b) 顶点 2 的内分支度为 1，外分支度为 2。  
 (c) 顶点 3 的外分支度为 2、内分支度为 1。

# 数据结构(C语言版)

责任编辑：张彦青

封面设计：杨月静

## 本书特色：

本书以 C 语言为程序设计语言，采用条列式的叙述方式引导读者循序渐进地学习各种数据结构，本书还使用图形来表示每一步的处理操作并配以浅显易懂的文字说明及范例程序，让读者可以轻松地了解数据结构的概念。

本书包括了：

- ◆ 以步骤的方式详细介绍各种概念
- ◆ 以百余个范例来搭配实作与应用
- ◆ 以百余个习题供读者练习与自我评估
- ◆ 用最详尽的时间复杂度分析计算
- ◆ 用最完整的图例说明各种结构
- ◆ 以浅显易懂的文字来解释概念
- ◆ 以简洁清晰的注释来辅助说明
- ◆ 以实际运行的程序来加深学习

ISBN 7-302-04509-7



9 787302 045090 >

定价：32.00 元



文魁资讯授权出版。  
限于中国大陆地区发行销售。